wacryptolib

Release 0.11

Pascal Chambon, Manon Michelet, Akram Bourichi, Francinette A

Mar 30, 2024

CONTENTS

1 Witness Angel Cryptolib **1**
1.1 Overview . . . 1
1.2 Installing the lib . . . 1
1.3 CLI interface . . . 1

2 Cryptolib concepts **3**
2.1 Different digital keys . . . 3
2.2 Data naming and integrity . . . 3
2.3 Keypair repositories . . . 4
2.4 Trusted parties . . . 4
2.5 Cryptainers and cryptoconfs . . . 5

3 Discover cryptainers **7**
3.1 Overview . . . 7
3.2 Encrypting data into a cryptainer . . . 8
3.3 Decrypting data from a cryptainer . . . 9
3.4 Noteworthy fields of a cryptainer . . . 10

4 Cryptoconf examples **11**
4.1 Simple cryptoconf . . . 11
4.2 Complex cryptoconf . . . 13

5 Flightbox CLI Tutorial **17**
5.1 CLI overview . . . 17
5.2 Playing with default encryption . . . 18
5.3 Creating an authenticator trustee . . . 20
5.4 Importing foreign keystores . . . 22
5.5 Generating simple cryptoconfs . . . 23
5.6 Securely encrypting data . . . 26
5.7 Managing cryptainers . . . 26

6 Flightbox CLI Reference **29**
6.1 flightbox . . . 29

7 Selected algorithms and formats **39**
7.1 Symmetric ciphers . . . 39
7.2 Asymmetric ciphers . . . 40
7.3 Signature algorithms . . . 40
7.4 Hashing functions . . . 40
7.5 Storage formats . . . 40
7.6 Communication protocols . . . 41

7.7 Notes on safety and performance 41

8 API Documentation **43**
8.1 Cryptainer 43
8.2 Trustee 48
8.3 Authenticator 49
8.4 Authentication device 50
8.5 Key generation 50
8.6 Key storage 52
8.7 Signature 58
8.8 Encryption 59
8.9 Shared secret 63
8.10 Sensor 63
8.11 Json-rpc client 65
8.12 Utilities 65
8.13 Exceptions 67
8.14 Error handling 68
8.15 Scaffolding for tests 70

9 Development instructions **71**
9.1 Getting started 71
9.2 Release process 72

10 Indices and tables **73**

Python Module Index **75**

Index **77**

CHAPTER

ONE

WITNESS ANGEL CRYPTOLIB

-> Full documentation on READTHEDOCS! <-

1.1 Overview

The Witness Angel Cryptolib is a toolkit aimed at handling secure configuration-driven containers, called *cryptainers*.

By leveraging a flexible JSON-based format called *cryptoconf*, users can define their own hybrid cryptosystem, recursively combining symmetric cihers, asymmetric ciphers, shared secrets, and data signatures.

Access to the cryptainers is secured by a variety of actors: local device, remote servers, trusted third parties...

The decryption process can involve different steps, like entering passphrases, or submitting authorization requests to remote "key guardians".

Overall, the lib gathers utilities to generate and store cryptographic keys, encrypt/check/decrypt cryptainers, access webservices and recorder sensors, and help testing other libraries willing to extend these tools.

1.2 Installing the lib

Just launch inside your python environment:

pip install wacryptolib

1.3 CLI interface

A command-line interface launcher, **flightbox**, is available to play with simple cryptainers.

```
$ flightbox --help
```

Look at the Flightbox manual, on readthedocs.org, for more details.

CHAPTER

TWO

CRYPTOLIB CONCEPTS

To avoid too long variable name, and to disambiguate words like "key" which have too many different meanings (symmetric key, asymmetric key, usb key, index key...), this library introduces its own set of terms, in addition to those already widely used in cryptography (cipher, signature, hash...).

2.1 Different digital keys

Lots of different "keys" and related concepts are used cryptolib code, so we use more precise terms when relevant.

A **key-algo** is a reference to either an encryption scheme (symmetric or asymmetric), or to a signature scheme (always asymmetric). In some parts of the code, when the purpose of the key is already sure, the names "cipher-algo" or "signature-algo" are used instead.

A **keypair** is a dict containing both "public" and "private" keys, for use in an asymmetric cipher or signature. Depending on the situation, these keys can be python objects, or serialized as PEM bytestrings (the private key being then possibly passphrase-protected. Keypairs are meant to be identified by a pair of **[keychain_uid, key_algo] references** (since for security, a keypair should indeed only be used for a single purpose). A *keychain* is thus of set of keypairs having a common UUID, but each used for a different purpose/algorithm.

A **symkey** is a dict containing all parameters required to configure a symmetric encryption/decryption cipher. Typically, it means a binary "secret key", and additional fields like "initialization vector" or "nonce", depending on the symmetric key-algo concerned. These symkeys are meant to be randomly generated, immediately protected by asymmetric ciphers, and stored along the encrypted data - as usual in a hybrid cryptosystem. Symkeys are anonymous.

When a bytestring (typically a serialized symkey) is split via a "Shamir shared secret" scheme, we refer to the different parts as **shards** (and not "shares", the other possible name). However, symkeys and their shards are often all called "symkeys" in the code, when the difference doesn't matter (e.g. when issuing decryption authorization requests).

Note that inside configurations and containers, we mostly use the term **key**, since the context should make it clear what exactly is at stake (mostly symmetric keys or their shards being encrypted via miscellaneous algorithms)

2.2 Data naming and integrity

We use **payload** to designate the actual content of recorded data (audio, video, gps...), at various stages of its encryption process.

We use **cleartext** and **ciphertext** to differentiate *BINARY* (not actual text) data before and after its encryption; although, since we're in a multi-layered encryption scheme, the ciphertext of an encryption layer becomes the cleartext for the next one.

The word **digest** is used, instead of "hash", to name the short bytestring obtained by hashing a payload. This digest can then be used as the "message" on which a timestamped signature is applied by a trustee, offering of proof of payload integrity and anteriority. The term **mac** (message authentication code) is used, instead of "tag", to designate a short bytestring which might be obtained at the end of a symmetric encryption operation. This bytestring offers of proof of the payload integrity, and also authenticity (i.e. it was well encrypted with the provided secret key). Those digests and macs can be considered all together as **integrity tags**.

Only when dealing with bytestrings that could be anything (serialized key, serialized json, final encrypted payload...), for example in filesystem utilies, we use the all-embracing term **data**.

2.3 Keypair repositories

A **keystore** is a generic storage for asymmetric keypairs. Its is typically a set of public/private PEM files in a directory. Keys can be a predetermined and immutable set, or on the contrary generated on demand; private keys can be present or not, protected by passphrases or not; a JSON metadata file describing this repository (type, owner, unique id...) can be present or not; it all depends on the subtype of the keystore.

The **local-keyfactory keystore** is the default keystore available on recording devices, as well as **server trustees** (see below). It can generate keypairs on demands, and typically doesn't protect them with passphrases.

An **authenticator** is a subtype of keystore used to provide a digital identity to a trusted third party (typically an individual). It is a fixed set of keypairs, all protected by the same passphrase, with some additional authentication fields in a metadata file. An **authdevice**, or authentication device, is a physical device on which an authenticator can be stored (for now, we use a simple folder at the root of an usb storage), if it is not simply stored in a local disk.

Authenticators can publish their public keys, and public metadata, to a **web gateway** - a simple online registry - so that other people may easily access them.

When keystores are **imported** from an authdevice or a web gateway, the imported copies naturally only contains a part (public, or at least without confidential information) of the initial authenticator.

2.4 Trusted parties

The data encrypted inside a cryptainer relies on multiple actors to protect itself from unwanted access.

Each of these actors is commonly designated as a **trustee** in the code. A trustee offers a set of standard services: providing public keys for encryption, delivering messages signatures, and treating requests for data decryption.

A recorder device has a **local-keyfactory trustee**, backed by the local-keyfactory keystore, which can encrypt and sign data using its own generated-on-demand digital keys.

But real protection is provided by trustees also called **key guardians**, which are *trusted third parties*. Access to these remote trustees is generally done via Internet, even if other channels (e.g. usb devices temporarily plugged in) can be used too. For now, these remote trustees can be **server trustees** (e.g. a database-backed keystore administrated by an association), or **authenticator trustees** (e.g. an individual having an authenticator keystore on his smartphone).

2.5 Cryptainers and cryptoconfs

For more information on these concepts, see the *dedicated page*.

CHAPTER

THREE

DISCOVER CRYPTAINERS

3.1 Overview

A **cryptainer** is a flexible, recursive structure dedicated to hybrid encryption. Each piece of data is encrypted one or several times in a row, and the "keys" (random symmetric keys, or shards obtained from a shared secret algorithm) used for these encryption operations become themselves pieces of encryptable data, thus repeating the process.

This gives a tree of protected data, in which the leaves are "trustees" relying on asymmetric (public key) encryption algorithms to protect intermediate "keys", and thus secure the encrypted payload.

This structure allows for some nice features:

- Fine-grained permission system: a complex mix of "mandatory" and "optional" trustees can protect the data together
- Auditability: chosen algorithms and other metadata are clearly exposed, allowing anyone to check that security level is sufficient
- Evolutive encryption: if some algorithms become unsafe, or if a storage wants to add an additional level of safety, it is easy to strengthen the cryptainer by adding some layers of encryption/signature in chosen parts of its tree.

The structure of a cryptainer is driven by a configuration tree called a **cryptoconf**.

A cryptoconf actually defines the *skeleton* of a cryptainer. During encryption, this structure will be filled with ciphertexts, integrity tags, signatures, and miscellaneous metadata to become the actual cryptainer.

Here is an example of medium-complexity cryptoconf:

```
  +----------+
  | payload |
  |cleartext|
  +----------+
       |
       |
[AES-EAX CIPHER] <-[RSA-OAEP CIPHER]- [KEY-GUARDIAN "witnessangel.com"]
       |
       v
 +------------+
 |  payload   | <-[RSA-PSS SIGNATURE]- [KEY-GUARDIAN: "Paul Dupond"]
 |ciphertext 1| <-[ECC_DSS SIGNATURE]- [KEY-GUARDIAN: Local Device]
 +------------+
       |                                         _
       |                                        /  <-[RSA-OAEP CIPHER- [KEY-GUARDIAN:
↪"John Doe"]
```

(continues on next page)

(continued from previous page)

```
[AES-CBC CIPHER] <-[SHARED-SECRET (2 on 3))-  -- <-[RSA-OAEP CIPHER- [KEY-GUARDIAN:
↪"Jane Doe"]
       |                                       \_ <-[RSA-OAEP CIPHER- [KEY-GUARDIAN:
↪"Jack Doe"]
       v
 +-------------+
 |  payload    |
 |ciphertext 2|
 +-------------+
```

In this custom cryptosystem:

- the sensitive payload is first encrypted by AES-EAX, then by AES-CBC.
- The randomly-generated symmetric keys used for these symmetric encryptions are then protected in different ways.
- The first one is encrypted with the public key of the key guardian server "witnessangel.com".
- The second one is split into 3 "shards", each protected by the public key of a different member of the family Doe.
- Two timestamped signatures are stapled to the cryptainer, both applied on the intermediate ciphertext (so they will only be verifiable after the outer AES-CBC layer has been decrypted)

3.2 Encrypting data into a cryptainer

3.2.1 Encryption/signature of the payload

The first "pipeline" of encryption has a special status, because it deals with the protection of the initial data (audio, video, or any other medium/document), called "payload" in our terminology.

This payload can have a very large size, and it is the actual, sensitive information to secure against reading and modification.

So for this first layer of encryption:

- The payload ciphertext can be stored apart from the cryptainer tree (we then call it an "offloaded payloaded")
- Only symmetric ciphers are allowed, since asymmetric ones are slow and often insecure when handling big inputs
- The resulting ciphertext can be signed and timestamped by one or more trustees (the initial payload can't be signed, for now, as this might leak information about its content)

3.2.2 Encryption of the keys

At each level of the rest of the recursive cryptainer tree, the currently considered "key" goes through its own pipeline of encryption. Each node of this pipeline receives as cleartext the ciphertext of the previous node, and can be one of one of these type:

- a shared secret: "splits" the data into N shards, with M of them required to reconstitute the data; each shard is then encrypted through its own pipeline
- a symmetric cipher: encrypts the data using a randomly generated key, which is then encrypted through its own pipeline

- an asymmetric cipher: encrypts the data using a public key owned by a "trustee"; this ends the recursion on that branch of the cryptainer

3.2.3 Different modes of encryption processing

If a payload of cleartext data is available, it can be encrypted and dumped to file in a single pass; however this can consume a lot of CPU and RAM on the moment.

As an alternative, the cryptolib supports on-the-fly encryption of data chunks transmitted during a long recording.

For that, we must setup a recording toolchain consisting of:

- sensors which get pushed, or at the contrary go pull, recorded chunks from hardware
- aggregators which combine data chunks into proper cleartext media/documents
- a pipeline which consumes cleartext data chunk by chunk, encrypts it, and streams it to disk (as an offloaded ciphertext)

In this scenario, the JSON cryptainer structure is initialized at the beginning of the recording, and remains in a *pending* state. At the end of the recording, integrity tags and payload signatures get added to this work-in-progress cryptainer, and it becomes complete.

3.3 Decrypting data from a cryptainer

Like in any layered encryption structure, decryption has to be performed from the outer shell to the core of the cryptainer.

This means that each key encryption pipeline is rolled back to recover a cleartext "key", which is in turn used to roll back the pipeline below it.

Along the way, payload integrity is verified thanks to both ciphertext signatures (checked via the public key of the related trustee), and integrity tags/macs (built in each symmetric or asymmetric cipher).

Since the leaves of the cryptainer tree are protected by trustees, they require external operations to be decrypted.

- "Local Key Factory" trustees are the easiest: their generated-on-demand keypairs have no passphrase protection on their private keys, so as long as these private keys are present (typically, on the recording device), decryption will succeed.
- "Server" trustees rely on keypairs generated-on-demand on a remote server (typically without passphrase protection of private keys). These trustees require decryption authorization requests to be submitted in advance to the server. When these permissions are then granted by an administrator, the server will accept to decrypt "key" ciphertexts submitted during the subsequent decryption operation.
- "Authenticator" trustees are individual key guardians having generated their own digital identity, with a set of keypairs protected by their (secret) passphrase. There are two ways to achieve decryption with them : either import their private keys locally and ask for their passphrase (low security), or send a secure key exchange request on a common web registry, which key guardians will then accept/reject from their own Authenticator device (high security).

Known limitations: As of today, the wacryptolib decryptor works in a single pass, and doesn't support partial decryption of cryptainers. It means that all "leaves" of the cryptainer tree must be unlocked in advance, by their relevant trustee. In practice, it means that all "Authenticator" trustees should be at the end of their "key" encryption pipeline, else they do not have access to the "key" ciphertext which must be sent as part of a decryption authorization request (so only the direct input of a passphrase would work). So instead of stacking 3 authenticator-backed RSA-OAEP encryptions in a row, for example, it is better to stack 3 symmetric ciphers (like AES-CBC or ChacCha20), and then protect each of their 3 randomly generated symkeys with a single authenticator-backed asymmetric encryption.

3.4 Noteworthy fields of a cryptainer

The *cryptainer_uid* field, located at the root of a cryptainer, uniquely identifies it.

The *keychain_uid* field, located nearby, can on the contrary be shared by several cryptainers, which thus end up targeting a common keychain of keypairs held by trustees (these keypairs being differentiated by their key-algo value). However this default *keychain_uid* can also be overridden deeper in the configuration tree, for each asymmetric cipher node.

A *metadata* dict field can be used to store miscellaneous information about the cryptainer (time, location, type of recording...).

CHAPTER

FOUR

CRYPTOCONF EXAMPLES

4.1 Simple cryptoconf

Below is a minimal cryptainer configuration in python format, with a single encryption layer and its single signature, both backed by the local "trustee" (or "key guardian") of the device; this workflow should not be used in real life of course, since the data is not protected against illegal reads.

```
{
  "payload_cipher_layers":[
    {
      "key_cipher_layers":[
        {
          "key_cipher_algo":"RSA_OAEP",
          "key_cipher_trustee":{
            "trustee_type":"local_keyfactory"
          }
        }
      ],
      "payload_cipher_algo":"AES_CBC",
      "payload_signatures":[
        {
          "payload_digest_algo":"SHA256",
          "payload_signature_algo":"DSA_DSS",
          "payload_signature_trustee":{
            "trustee_type":"local_keyfactory"
          }
        }
      ]
    }
  ]
}
```

A corresponding cryptainer content, in Pymongo's Extended Json format (base64 bytestrings shortened for clarity), looks like this. Binary subType 03 means "UUID", whereas subType 00 means "bytes".

```
{
  "cryptainer_format":"cryptainer_1.0",
  "cryptainer_metadata":null,
  "cryptainer_state":"FINISHED",
  "cryptainer_uid":{
    "$binary":{
```

(continues on next page)

(continued from previous page)

```
      "base64":"Du14m64eb4m/+/uCPAkEqw==",
      "subType":"03"
    }
  },
  "keychain_uid":{
    "$binary":{
      "base64":"Du14m64emE23Dnuw4+aKFA==",
      "subType":"03"
    }
  },
  "payload_cipher_layers":[
    {
      "key_cipher_layers":[
        {
          "key_cipher_algo":"RSA_OAEP",
          "key_cipher_trustee":{
            "trustee_type":"local_keyfactory"
          }
        }
      ],
      "key_ciphertext":{
        "$binary":{
          "base64":"eyJkaWdlc3Rfb...JzdWJUeXBlIjogIjAwIn19XX0=",
          "subType":"00"
        }
      },
      "payload_cipher_algo":"AES_CBC",
      "payload_macs":{
      },
      "payload_signatures":[
        {
          "payload_digest_value":{
            "$binary":{
              "base64":"XgNeHINsXw16Tl...WtknjGh93nMB4v09Y=",
              "subType":"00"
            }
          },
          "payload_digest_algo":"SHA256",
          "payload_signature_algo":"DSA_DSS",
          "payload_signature_struct":{
            "signature_timestamp_utc":{
              "$numberInt":"1641305798"
            },
            "signature_value":{
              "$binary":{
                "base64":"F/q+FZQThx1JnyUCwwh...59NCRreWpf2BK8673qMc=",
                "subType":"00"
              }
            }
          },
          "payload_signature_trustee":{
            "trustee_type":"local_keyfactory"
```

(continues on next page)

(continued from previous page)

```
          }
        }
      ]
    }
  ],
  "payload_ciphertext_struct":{
    "ciphertext_location":"inline",
    "ciphertext_value":{
      "$binary":{
        "base64":"+6CAsNlLHTHFxVcw6M9p/SK...axRM3poryDA/BP9tBeaFU4Y=",
        "subType":"00"
      }
    }
  }
}
```

4.2 Complex cryptoconf

Below is a python data tree showing all the types of node possible in a cryptoconf.

We see the 3 currently supported types of trustee: *local_keyfactory*, *authenticator* (with a keystore_uid), and *jsonrpc_api* (with a jsonrpc_url).

We also see how share secrets, symmetric ciphers, and asymmetric ciphers (RSA_OAEP and its attached trustee) can be combined to create a deeply nested structure.

```
{
  "payload_cipher_layers":[
    {
      "key_cipher_layers":[
        {
          "key_cipher_algo":"RSA_OAEP",
          "key_cipher_trustee":{
            "jsonrpc_url":"http://www.mydomain.com/json",
            "trustee_type":"jsonrpc_api"
          }
        }
      ],
      "payload_cipher_algo":"AES_EAX",
      "payload_signatures":[
      ]
    },
    {
      "key_cipher_layers":[
        {
          "key_cipher_algo":"RSA_OAEP",
          "key_cipher_trustee":{
            "keystore_uid":UUID("320b35bb-e735-4f6a-a4b2-ada124e30190"),
            "trustee_type":"authenticator"
          }
        }
```

(continues on next page)

(continued from previous page)

```
      ],
      "payload_cipher_algo":"AES_CBC",
      "payload_signatures":[
        {
          "payload_digest_algo":"SHA3_512",
          "payload_signature_algo":"DSA_DSS",
          "payload_signature_trustee":{
            "trustee_type":"local_keyfactory"
          }
        }
      ]
    },
    {
      "key_cipher_layers":[
        {
          "key_cipher_algo":"[SHARED_SECRET]",
          "key_shared_secret_shards":[
            {
              "key_cipher_layers":[
                {
                  "key_cipher_algo":"RSA_OAEP",
                  "key_cipher_trustee":{
                    "trustee_type":"local_keyfactory"
                  }
                },
                {
                  "key_cipher_algo":"RSA_OAEP",
                  "key_cipher_trustee":{
                    "trustee_type":"local_keyfactory"
                  }
                }
              ]
            },
            {
              "key_cipher_layers":[
                {
                  "key_cipher_algo":"AES_CBC",
                  "key_cipher_layers":[
                    {
                      "key_cipher_algo":"[SHARED_SECRET]",
                      "key_shared_secret_shards":[
                        {
                          "key_cipher_layers":[
                            {
                              "key_cipher_algo":"RSA_OAEP",
                              "key_cipher_trustee":{
                                "trustee_type":"local_keyfactory"
                              },
                              "keychain_uid":UUID("65dbbe4f-0bd5-4083-a274-3c76efeecccc")
                            }
                          ]
                        }
```

(continues on next page)

(continued from previous page)

```
                    ],
                    "key_shared_secret_threshold":1
                  },
                  {
                    "key_cipher_algo":"RSA_OAEP",
                    "key_cipher_trustee":{
                      "trustee_type":"local_keyfactory"
                    }
                  }
                ]
              }
            ]
          },
          {
            "key_cipher_layers":[
              {
                "key_cipher_algo":"RSA_OAEP",
                "key_cipher_trustee":{
                  "trustee_type":"local_keyfactory"
                }
              }
            ]
          },
          {
            "key_cipher_layers":[
              {
                "key_cipher_algo":"RSA_OAEP",
                "key_cipher_trustee":{
                  "trustee_type":"local_keyfactory"
                },
                "keychain_uid":UUID("65dbbe4f-0bd5-4083-a274-3c76efeebbbb")
              }
            ]
          }
        ],
        "key_shared_secret_threshold":2
      }
    ],
    "payload_cipher_algo":"CHACHA20_POLY1305",
    "payload_signatures":[
      {
        "keychain_uid":UUID("0e8e861e-f0f7-e54b-18ea-34798d5daaaa"),
        "payload_digest_algo":"SHA3_256",
        "payload_signature_algo":"RSA_PSS",
        "payload_signature_trustee":{
          "trustee_type":"local_keyfactory"
        }
      },
      {
        "payload_digest_algo":"SHA512",
        "payload_signature_algo":"ECC_DSS",
        "payload_signature_trustee":{
```

(continues on next page)

(continued from previous page)

```
            "trustee_type":"local_keyfactory"
          }
        }
      ]
    }
  ]
}
```

Here is a summary of the same cryptoconf, as returned for example by the CLI "summarize" command.

```
Data encryption layer 1: AES_EAX
  Key encryption layers:
    RSA_OAEP via trustee 'server www.mydomain.com'
  Signatures: None
Data encryption layer 2: AES_CBC
  Key encryption layers:
    RSA_OAEP via trustee 'authenticator 320b35bb-e735-4f6a-a4b2-ada124e30190'
  Signatures:
    SHA3_512/DSA_DSS via trustee 'local device'
Data encryption layer 3: CHACHA20_POLY1305
  Key encryption layers:
    Shared secret with threshold 2:
      Shard 1 encryption layers:
        RSA_OAEP via trustee 'local device'
        RSA_OAEP via trustee 'local device'
      Shard 2 encryption layers:
        AES_CBC with subkey encryption layers:
          Shared secret with threshold 1:
            Shard 1:
              RSA_OAEP via trustee 'local device'
          RSA_OAEP via trustee 'local device'
      Shard 3 encryption layers:
        RSA_OAEP via trustee 'local device'
      Shard 4 encryption layers:
        RSA_OAEP via trustee 'local device'
  Signatures:
    SHA3_256/RSA_PSS via trustee 'local device'
    SHA512/ECC_DSS via trustee 'local device'
```

CHAPTER

FIVE

FLIGHTBOX CLI TUTORIAL

5.1 CLI overview

The flightbox command-line utility provides subcommands for end-to-end workflows around cryptainers.

```
$ flightbox -h
Usage: flightbox [OPTIONS] COMMAND [ARGS]...

  Flexible cryptographic toolkit for multi-tenant encryption and signature

Options:
  -v, --verbosity LVL             Either CRITICAL, ERROR, WARNING, INFO or
                                  DEBUG
  -k, --keystore-pool DIRECTORY   Folder tree to store keystores (else
                                  ~/.witnessangel/keystore_pool is used)
  -c, --cryptainer-storage DIRECTORY
                                  Folder to store cryptainers (else
                                  ~/.witnessangel/cryptainers is used)
  -g, --gateway-url TEXT          URL of the web registry endpoint
  -h, --help                      Show this message and exit.

Commands:
  authenticator     Manage authenticator trustees
  cryptainer        Manage encrypted containers
  cryptoconf        Manage cryptographic configurations
  encrypt           Turn a media file into a secure container
  foreign-keystore  Manage locally imported keystores
```

Note the keystore-pool and cryptainer-storage options: the first one is used to store local and imported keystores, and the second one is used to store encrypted containers.

Hint: The positioning of CLI options is rather strict: they must be added directly after the command they are related too, and before the following subcommand. So for example *--gateway-url* must be provided after *flightbox*, but before subcommands like *cryptainer*, *encrypt* etc.

5.2 Playing with default encryption

Imagine that we won't to encrypt a readme file. The corresponding command could be as simple as:

```
$ flightbox encrypt readme.rst
warning: No cryptoconf provided, defaulting to simple and INSECURE example cryptoconf
Data file 'readme.rst' successfully encrypted into storage cryptainer
```

But as the output mentions, this is not a satisfying encryption. Let's see why.

The encrypted container has well been saved into the Cryptainer Storage:

```
$ flightbox cryptainer list
+-------------------+--------+-----------+------------------+
|       Name        |  Size  | Offloaded | Created at (UTC) |
+-------------------+--------+-----------+------------------+
| readme.rst.crypt  | 102 KB |     X     | 2024-03-30 21:15 |
+-------------------+--------+-----------+------------------+
```

Hint: For performance reasons, the cryptainer is, by default, "offloaded". This means it is separated in two files: the metadata in "readme.rst.crypt", and the (possibly huge) encrypted data in "readme.rst.crypt.payload".

If you open readme.rst.crypt with a text editor, you'll notice that it's just a JSON file, but in Pymongo's Extended Json format: it uses specific fields like $binary or $date to add better types to the serialized data.

Let's now check the structure of this cryptainer:

```
$ flightbox cryptainer summarize readme.rst.crypt
Loading cryptainer readme.rst.crypt from storage (include_payload_ciphertext=True)

Data encryption layer 1: AES_CBC
  Key encryption layers:
    RSA_OAEP via trustee 'local device'
    Shared secret with threshold 1:
      Shard 1 encryption layers:
        RSA_OAEP via trustee 'local device'
      Shard 2 encryption layers:
        RSA_OAEP via trustee 'local device'
  Signatures:
    SHA256/DSA_DSS via trustee 'local device'
```

As we see, this cryptainer uses several types of encryption, but only relies on autogenerated local-device keys, which are not protected by a passphrase.

This means that we can directly decrypt the content of this cryptainer:

```
$ flightbox cryptainer decrypt readme.rst.crypt -o readme.rst.decrypted
Loading cryptainer readme.rst.crypt from storage (include_payload_ciphertext=True)
Decryption report:
[{'entry_criticity': 'INFO',
  'entry_exception': None,
  'entry_message': 'Skipping retrieval of remotely predecrypted symkeys '
                   '(requires requestor-uid and gateway urls)',
```

(continues on next page)

(continued from previous page)

```
  'entry_nesting': 0,
  'entry_type': 'INFORMATION'},
 {'entry_criticity': 'INFO',
  'entry_exception': None,
  'entry_message': 'Starting decryption of payload cipher layer 1/1 (algo: '
                   'AES_CBC)',
  'entry_nesting': 0,
  'entry_type': 'INFORMATION'},
 {'entry_criticity': 'INFO',
  'entry_exception': None,
  'entry_message': 'Decrypting AES_CBC symmetric key through 2 cipher layer(s)',
  'entry_nesting': 1,
  'entry_type': 'INFORMATION'},
 {'entry_criticity': 'INFO',
  'entry_exception': None,
  'entry_message': 'Deciphering 2 shards of shared secret (threshold: 1)',
  'entry_nesting': 1,
  'entry_type': 'INFORMATION'},
 {'entry_criticity': 'INFO',
  'entry_exception': None,
  'entry_message': 'Decrypting shard #1 through 1 cipher layer(s)',
  'entry_nesting': 2,
  'entry_type': 'INFORMATION'},
 {'entry_criticity': 'INFO',
  'entry_exception': None,
  'entry_message': 'Attempting to decrypt key with asymmetric algorithm '
                   'RSA_OAEP (trustee: local_keyfactory)',
  'entry_nesting': 2,
  'entry_type': 'INFORMATION'},
 {'entry_criticity': 'INFO',
  'entry_exception': None,
  'entry_message': 'No predecrypted symmetric found (e.g. coming from remote '
                   'trustee)',
  'entry_nesting': 3,
  'entry_type': 'INFORMATION'},
 {'entry_criticity': 'INFO',
  'entry_exception': None,
  'entry_message': 'Attempting actual decryption of key using asymmetric '
                   'algorithm RSA_OAEP (via trustee)',
  'entry_nesting': 3,
  'entry_type': 'INFORMATION'},
 {'entry_criticity': 'INFO',
  'entry_exception': None,
  'entry_message': 'A sufficient number of shared-secret shards (1) have been '
                   'decrypted',
  'entry_nesting': 2,
  'entry_type': 'INFORMATION'},
 {'entry_criticity': 'INFO',
  'entry_exception': None,
  'entry_message': 'Attempting to decrypt key with asymmetric algorithm '
                   'RSA_OAEP (trustee: local_keyfactory)',
  'entry_nesting': 1,
```

(continues on next page)

(continued from previous page)

```
  'entry_type': 'INFORMATION'},
 {'entry_criticity': 'INFO',
  'entry_exception': None,
  'entry_message': 'No predecrypted symmetric found (e.g. coming from remote '
                   'trustee)',
  'entry_nesting': 2,
  'entry_type': 'INFORMATION'},
 {'entry_criticity': 'INFO',
  'entry_exception': None,
  'entry_message': 'Attempting actual decryption of key using asymmetric '
                   'algorithm RSA_OAEP (via trustee)',
  'entry_nesting': 2,
  'entry_type': 'INFORMATION'}]
Decryption of cryptainer 'readme.rst.crypt' to file 'readme.rst.decrypted' successfully␣
↪finished
```

We can then check that readme.rst and readme.rst.decrypted are well the same.

5.3 Creating an authenticator trustee

To encrypt data in a more secure fashion, we'll need some key guardians, called *trustees* in Flightbox.

The simplest form of trustee is an authenticator, a digital identity for a single person. Currently, it is backed by a keystore folder containing some metadata and a bunch of keypairs - all protected by the same "passphrase" (a very long password).

The standard way of generating this identity would be to use a standalone program like the mobile application Witness Angel Authenticator (for Android and iOS), and then to publish the public part of this identity to a web registry.

But we can also create authenticators via the CLI:

```
$ flightbox authenticator create ./mysphinxdocauthenticator --owner "John Doe" --
↪passphrase-hint "Some hint"
Using passphrase specified as WA_PASSPHRASE environment variable
Authenticator successfully initialized with 3 keypairs in directory /home/docs/checkouts/
↪readthedocs.org/user_builds/witness-angel-cryptolib/checkouts/latest/docs/
↪mysphinxdocauthenticator
```

For the needs of this doc generation, we had to provide the passphrase as an environment variable, but normally the program will just prompt the user for it.

We can then review the just-created authenticator:

```
$ flightbox authenticator view ./mysphinxdocauthenticator
+----------------------------+-----------------------------------------------+
|          Property          |                     Value                     |
+----------------------------+-----------------------------------------------+
|        keystore_uid        |      0f91ac29-22bf-d72b-1430-9240d700f928     |
|       keystore_owner       |                    John Doe                   |
|  keystore_passphrase_hint  |                   Some hint                   |
| keystore_creation_datetime |                2024-03-30 21:15               |
|    keypair_identifiers     | RSA_OAEP 0f91ac29-3d7b-41b6-64f1-7633e3bfc8a0 |
|                            | RSA_OAEP 0f91ac29-5ea1-a478-f5e3-7b698c66b40d |
```

(continues on next page)

(continued from previous page)

```
|                            | RSA_OAEP 0f91ac29-d532-b18d-d2e5-8ac4ef902587 |
+----------------------------+-----------------------------------------------+
```

Like for most list/view commands, we can switch to JSON format for automation purposes:

```
$ flightbox authenticator view ./mysphinxdocauthenticator --format json
{
  "keypair_identifiers": [
    {
      "key_algo": "RSA_OAEP",
      "keychain_uid": {
        "$binary": {
          "base64": "D5GsKT17QbZk8XYz47/IoA==",
          "subType": "04"
        }
      },
      "private_key_present": true
    },
    {
      "key_algo": "RSA_OAEP",
      "keychain_uid": {
        "$binary": {
          "base64": "D5GsKV6hpHj143tpjGa0DQ==",
          "subType": "04"
        }
      },
      "private_key_present": true
    },
    {
      "key_algo": "RSA_OAEP",
      "keychain_uid": {
        "$binary": {
          "base64": "D5GsKdUysY3S5YrE75Alhw==",
          "subType": "04"
        }
      },
      "private_key_present": true
    }
  ],
  "keystore_creation_datetime": {
    "$date": {
      "$numberLong": "1711833305158"
    }
  },
  "keystore_owner": "John Doe",
  "keystore_passphrase_hint": "Some hint",
  "keystore_uid": {
    "$binary": {
      "base64": "D5GsKSK/1ysUMJJA1wD5KA==",
      "subType": "04"
    }
  }
```

(continues on next page)

(continued from previous page)

```
}
```

We can check, later, that we still remember the right passphrase for this authenticator:

```
$ flightbox authenticator validate ./mysphinxdocauthenticator
Using passphrase specified as WA_PASSPHRASE environment variable
Authenticator has UID 0f91ac29-22bf-d72b-1430-9240d700f928 and belongs to owner John Doe
Creation date: 2024-03-30 21:15:05+00:00
Keypair count: 3

Authenticator successfully checked, no integrity errors found
```

We can delete the authenticator with `flightbox authenticator delete ./mysphinxdocauthenticator`, which is the same as manually deleting the folder.

5.4 Importing foreign keystores

Authenticators are supposed to be remote identities, well protected by their owner. To use them in our encryption system, we need to import their public keys, which are like "padlocks". That's what we call "foreign keystores" - partial local copies of remote identities.

Let's begin by importing the authenticator we just created.

```
$ flightbox foreign-keystore import --from-path ./mysphinxdocauthenticator
Importing foreign keystore from folder /home/docs/checkouts/readthedocs.org/user_builds/
↪witness-angel-cryptolib/checkouts/latest/docs/mysphinxdocauthenticator, without␣
↪private keys
Authenticator 0f91ac29-22bf-d72b-1430-9240d700f928 (owner: John Doe) imported
```

Let's also import an identity from a web registry, using its UUID that the owner gave us directly.

```
$ flightbox --gateway-url https://api.witnessangel.com/gateway/jsonrpc/ foreign-keystore␣
↪import --from-gateway 0f0c0988-80c1-9362-11c1-b06909a3a53c
Importing foreign keystore 0f0c0988-80c1-9362-11c1-b06909a3a53c from web gateway
Initiating remote call 'get_public_authenticator()' to server https://api.witnessangel.
↪com/gateway/jsonrpc/
Authenticator 0f0c0988-80c1-9362-11c1-b06909a3a53c (owner: ¤aaa) imported
```

If we have setup authenticators in default locations of connected USB keys, we can automatically import them:

```
$ flightbox foreign-keystore import --from-usb --include-private-keys
Importing foreign keystores from USB devices, with private keys
0 new authenticators imported, 0 updated, 0 skipped because corrupted
```

Warning: The *--include-private-keys* option requests that the private part of the identity be imported too, if present (which is not the case e.g. for web gateway identities). This is only useful if one intends to decrypt data locally, by entering passphrases during decryption. But much more secure workflows are now available, for example by using the mobile application Authenticator.

We can then review the imported keystores, which will be usable for encryption:

```
$ flightbox foreign-keystore list
+--------------------------------------+----------+-------------+--------------+---------
↪---------+
|             Keystore UID             |  Owner   | Public keys | Private Keys | Created␣
↪at (UTC) |
+--------------------------------------+----------+-------------+--------------+---------
↪---------+
| 0f0c0988-80c1-9362-11c1-b06909a3a53c |   ¤aaa   |      7      |      0       |        ␣
↪         |
| 0f91ac29-22bf-d72b-1430-9240d700f928 | John Doe |      3      |      0       | 2024-03-
↪30 21:15 |
+--------------------------------------+----------+-------------+--------------+---------
↪---------+
```

And we can check the keypairs present in a specific keystore, this way:

```
$ flightbox foreign-keystore view 0f0c0988-80c1-9362-11c1-b06909a3a53c
+----------------------------+-----------------------------------------------+
|          Property          |                     Value                     |
+----------------------------+-----------------------------------------------+
|        keystore_uid        |      0f0c0988-80c1-9362-11c1-b06909a3a53c     |
|       keystore_owner       |                      ¤aaa                     |
| keystore_creation_datetime |                                               |
|    keypair_identifiers     | RSA_OAEP 0f0c0989-1111-a226-c471-99cbb2d203c3 |
|                            | RSA_OAEP 0f0c0989-a4fa-7c15-0b07-932729d9dc5b |
|                            | RSA_OAEP 0f0c0989-e6ae-f533-d92a-cc2dc651acb8 |
|                            | RSA_OAEP 0f0c098b-5d2f-633c-167b-a8d5c86067ff |
|                            | RSA_OAEP 0f0c098c-47a6-6a2e-7071-f28f97c3093a |
|                            | RSA_OAEP 0f0c098c-ce02-7828-a519-45e6df84a25f |
|                            | RSA_OAEP 0f0c098d-59a3-1061-92a0-fcd9549a7dac |
+----------------------------+-----------------------------------------------+
```

We can later delete the foreign keystore with `flightbox foreign-keystore delete 0f0c0988-80c1-9362-11c1-b06909a3a53c`, which is the same as manually deleting the folder deep inside the keystore pool.

5.5 Generating simple cryptoconfs

Now that we have locally registered some trustees, it's time to specify how they should protect our data, how they should become our "key guardians". This happens with a cryptoconf, a JSON cryptainer template recursively describing the different layers of encryption to be used on data and on keys, as well as the signatures to apply.

Cryptoconf can be very complex; but for some low-depth, signatureless cases, we can use the CLI to generate a cryptoconf for us.

For example, imagine we want to encrypt the data using the AES-CBC cipher, and then protect the (random) secret key of this cipher using a keypair of the trustee imported from the web gateway.

```
$ flightbox cryptoconf generate-simple add-payload-cipher-layer --sym-cipher-algo AES_
↪CBC add-key-cipher-layer --asym-cipher-algo RSA_OAEP --trustee-type authenticator --
↪keystore-uid 0f0c0988-80c1-9362-11c1-b06909a3a53c --keychain-uid 0f0c0989-1111-a226-
↪c471-99cbb2d203c3
```

(continues on next page)

(continued from previous page)

```
{
  "payload_cipher_layers": [
    {
      "key_cipher_layers": [
        {
          "key_cipher_algo": "RSA_OAEP",
          "key_cipher_trustee": {
            "keystore_uid": {
              "$binary": {
                "base64": "DwwJiIDBk2IRwbBpCaOlPA==",
                "subType": "04"
              }
            },
            "trustee_type": "authenticator"
          },
          "keychain_uid": {
            "$binary": {
              "base64": "DwwJiREROibEcZnLstIDww==",
              "subType": "04"
            }
          }
        }
      ],
      "payload_cipher_algo": "AES_CBC",
      "payload_signatures": []
    }
  ]
}
```

The UUIDs that we selected are well there, even if unrecognizable in the $binary/base64 format of the JSON.

Hint: What are the keychain UIDs added above?

They uniquely identify a keypair for a given algorithm and trustee.

Each cryptainer has a root keychain UID, autogenerated if not provided by the cryptoconf. This default keychain UID is sufficient for the local-keyfactory trustee, since it will generate new keypairs on demand. But for each authenticator trustees, the set of available keypairs is already determined; so we must choose the keypair that we want to rely on, by providing its keychain UID at trustee level.

We can go farther, and decide that we want two layers of data encryption:

- one protected by an autogenerated local key
- the other protected by a shared secret between two authenticators, any of these two being sufficient to decrypt the data.

Here is how such a configuration could be generated:

```
$ flightbox cryptoconf generate-simple
    add-payload-cipher-layer --sym-cipher-algo AES_CBC
        add-key-cipher-layer --asym-cipher-algo RSA_OAEP --trustee-type local_keyfactory
    add-payload-cipher-layer --sym-cipher-algo CHACHA20_POLY1305
        add-key-shared-secret --threshold 1
```

(continues on next page)

(continued from previous page)

```
            add-key-shard --asym-cipher-algo RSA_OAEP --trustee-type authenticator --
↪keystore-uid 0f0c0988-80c1-9362-11c1-b06909a3a53c --keychain-uid 0f0c0989-1111-a226-
↪c471-99cbb2d203c3
            add-key-shard --asym-cipher-algo RSA_OAEP --trustee-type authenticator --
↪keystore-uid 7a25db2c-4c4e-42bb-a064-8da2007a4fd7 --keychain-uid 8c57e283-308a-4c78-
↪86f9-ee6176757a6f
    > shared-secret-cryptoconf.json``
```

And here is the resulting cryptoconf structure:

```
$ flightbox cryptoconf summarize sophisticated-cryptoconf.json

Data encryption layer 1: AES_CBC
  Key encryption layers:
    RSA_OAEP via trustee 'local device'
  Signatures: None
Data encryption layer 2: CHACHA20_POLY1305
  Key encryption layers:
    Shared secret with threshold 1:
      Shard 1 encryption layers:
        RSA_OAEP via trustee 'authenticator 0f0c0988-80c1-9362-11c1-b06909a3a53c'
      Shard 2 encryption layers:
        RSA_OAEP via trustee 'authenticator 7a25db2c-4c4e-42bb-a064-8da2007a4fd7'
  Signatures: None
```

If we want a logical AND instead of a logical OR between the two authenticator-based trustees, either we increase the *threshold* to 2, or we apply the trustee protections one after the other, like this:

```
$ flightbox cryptoconf generate-simple add-payload-cipher-layer --sym-cipher-algo AES_CBC
    add-key-cipher-layer --asym-cipher-algo RSA_OAEP --trustee-type authenticator --
↪keystore-uid 0f0c0988-80c1-9362-11c1-b06909a3a53c --keychain-uid 0f0c0989-1111-a226-
↪c471-99cbb2d203c3 --sym-cipher-algo AES_EAX
    add-key-cipher-layer --asym-cipher-algo RSA_OAEP --trustee-type authenticator --
↪keystore-uid 7a25db2c-4c4e-42bb-a064-8da2007a4fd7 --keychain-uid 8c57e283-308a-4c78-
↪86f9-ee6176757a6f
    > multikeylayer-cryptoconf.json
```

Which gives this structure:

```
$ flightbox cryptoconf summarize multikeylayer-cryptoconf.json

Data encryption layer 1: AES_CBC
  Key encryption layers:
    AES_EAX with subkey encryption layers:
      RSA_OAEP via trustee 'authenticator 0f0c0988-80c1-9362-11c1-b06909a3a53c'
    RSA_OAEP via trustee 'authenticator 7a25db2c-4c4e-42bb-a064-8da2007a4fd7'
  Signatures: None
```

Thus, the randomly generated AES-CBC is secured by the first trustee, and then the result of this encryption is fed to the second trustee, which secures it too.

Hint: Note that we used an hybrid encryption (AES-EAX/RSA-OAEP) for the first layer of key encryption; this is not mandatory, but it avoid stacking trustees directly one over the other in these "Key encryption layers".

When trustees are directly stacked, decryption is complicated because we must decrypt the Key through the upper layer, before being able to query the next trustee for authorization, using the now partially-decrypted Key.

When trustees are separated "leaves" of the cryptoconf/cryptainer tree, on the contrary, they can all be queried in parallel for authorizations, each one being fed its corresponding encrypted "Key" (here, respectively the encrypted AES-CBC and AES-EAX keys).

Note that a `flightbox cryptoconf validate <file>` command is available, to check JSON cryptoconfs that you have generated by other means.

We haven't evocated, in this tutorial, the **"server-backed" trustees** (trustee_type "jsonrpc_api" in cryptoconfs). They can be used as encryption/signature keypair providers, but the recorder device must be constantly connected to Internet, and their current decryption workflow offers low security, so we'd rather not use them for now.

5.6 Securely encrypting data

Now that we have dealt with trustees and cryptoconf, the rest is easy:

```
$ flightbox encrypt readme.rst --cryptoconf cryptoconf.json -o readme
Data file 'readme' successfully encrypted into storage cryptainer
```

Notice that we can choose the basename of the target cryptainer with `-o`. There is also a `--bundle` option to output the cryptainer as a single file - handy if the input file is rather enough.

5.7 Managing cryptainers

The commands to list, summarize, validate, and delete cryptainers from the current "Cryptainer Storage" are quite straightforward:

```
$ flightbox cryptainer -h
Usage: flightbox cryptainer [OPTIONS] COMMAND [ARGS]...

  Manage encrypted containers

Options:
  -h, --help  Show this message and exit.

Commands:
  decrypt    Turn a cryptainer back into the original media file
  delete     Delete a local cryptainer
  list       List local cryptainers
  purge      Delete oldest cryptainers per criteria
  summarize  Display a summary of a cryptainer structure
  validate   Validate a cryptainer structure
```

The `purge` command can combine multiple criteria to ensure that technical and legal constraints are met. For example if we can only keep the cryptainers 30 days, and want to limit their count to 100 and their total space to 1000 MB, we can run:

```
$ flightbox cryptainer purge --max-age 30 --max-count 100 --max-quota 1000
Intentionally purging cryptainers
Cryptainers successfully deleted: 0
```

Finally, the `decrypt` command is not relevant for our new cryptoconfs, since it doesn't support the complex mix of passphrases and remote authorization requests necessary to reveal a Flightbox cryptainer. It's better to use some Revelation Station software, like that included in the **W.A Recorder** program of Witness Angel project.

CHAPTER

SIX

FLIGHTBOX CLI REFERENCE

6.1 flightbox

Flexible cryptographic toolkit for multi-tenant encryption and signature

```
flightbox [OPTIONS] COMMAND [ARGS]...
```

Options

-v, --verbosity <LVL>
Either CRITICAL, ERROR, WARNING, INFO or DEBUG

-k, --keystore-pool <keystore_pool>
Folder tree to store keystores (else ~/.witnessangel/keystore_pool is used)

-c, --cryptainer-storage <cryptainer_storage>
Folder to store cryptainers (else ~/.witnessangel/cryptainers is used)

-g, --gateway-url <gateway_url>
URL of the web registry endpoint

6.1.1 authenticator

Manage authenticator trustees

```
flightbox authenticator [OPTIONS] COMMAND [ARGS]...
```

create

Initialize an authenticator folder with a set of keypairs

The target directory must not exist yet, but its parent directory must exist.

Authenticator passphrase can be provided as WA_PASSPHRASE environment variable, else user will be prompted for it.

No constraints are applied to the lengths of the passphrase or other fields, so beware of security considerations!

```
flightbox authenticator create [OPTIONS] AUTHENTICATOR_DIR
```

Options

--keypair-count <keypair_count>

Count of keypairs to generate (min 1)

Default
`3`

--owner <owner>

Required Name of the authenticator owner

--passphrase-hint <passphrase_hint>

Required Non-sensitive hint to help remember the passphrase

Arguments

AUTHENTICATOR_DIR

Required argument

delete

Delete an authenticator folder along with all its content

```
flightbox authenticator delete [OPTIONS] AUTHENTICATOR_DIR
```

Arguments

AUTHENTICATOR_DIR

Required argument

validate

Verify the metadata and keypairs of an authenticator folder

Authenticator passphrase can be provided as WA_PASSPHRASE environment variable, else user will be prompted for it.

```
flightbox authenticator validate [OPTIONS] AUTHENTICATOR_DIR
```

Arguments

AUTHENTICATOR_DIR

Required argument

view

View metadata and public keypair identifiers of an authenticator

The presence and validity of private keys isn't checked.

```
flightbox authenticator view [OPTIONS] AUTHENTICATOR_DIR
```

Options

-f, --format <format>

> **Default**
> `plain`
>
> **Options**
> plain | json

Arguments

AUTHENTICATOR_DIR

Required argument

6.1.2 cryptainer

Manage encrypted containers

```
flightbox cryptainer [OPTIONS] COMMAND [ARGS]...
```

decrypt

Turn a cryptainer back into the original media file

This command is for test purposes only, since it only works with INSECURE cryptoconfs where private keys are locally available, and not protected by passphrases.

For real world use cases, see the Witness Angel software suite (Authenticator, Revelation Station...).

```
flightbox cryptainer decrypt [OPTIONS] CRYPTAINER_NAME
```

Options

-o, --output-file <output_file>

Arguments

CRYPTAINER_NAME

Required argument

delete

Delete a local cryptainer

```
flightbox cryptainer delete [OPTIONS] CRYPTAINER_NAME
```

Arguments

CRYPTAINER_NAME

Required argument

list

List local cryptainers

```
flightbox cryptainer list [OPTIONS]
```

Options

-f, --format <format>

Default
plain

Options
plain | json

purge

Delete oldest cryptainers per criteria

```
flightbox cryptainer purge [OPTIONS]
```

Options

--max-age <max_age>

Maximum age of cryptainer, in days

--max-count <max_count>

Maximum count of cryptainers in storage

--max-quota <max_quota>

Maximum total size of cryptainers, in MBs

summarize

Display a summary of a cryptainer structure

```
flightbox cryptainer summarize [OPTIONS] CRYPTAINER_NAME
```

Arguments

CRYPTAINER_NAME
Required argument

validate

Validate a cryptainer structure

```
flightbox cryptainer validate [OPTIONS] CRYPTAINER_NAME
```

Arguments

CRYPTAINER_NAME
Required argument

6.1.3 cryptoconf

Manage cryptographic configurations

```
flightbox cryptoconf [OPTIONS] COMMAND [ARGS]...
```

generate-simple

Generate a simple cryptoconf using subcommands

```
flightbox cryptoconf generate-simple [OPTIONS] COMMAND1 [ARGS]... [COMMAND2
                                     [ARGS]...]...
```

Options

--keychain-uid <keychain_uid>
Default UID for asymmetric keys

add-key-cipher-layer

Add a layer of asymmetric encryption of the key

A symmetric cipher can also be used, resulting in a hybrid encryption scheme.

```
flightbox cryptoconf generate-simple add-key-cipher-layer [OPTIONS]
```

Options

--asym-cipher-algo <asym_cipher_algo>

Required Asymmetric algorithms for key encryption

Options
RSA_OAEP

--trustee-type <trustee_type>

Required Kind of key-guardian used

Options
local_keyfactory | authenticator

--keystore-uid <keystore_uid>

UID of the key-guardian (only for authenticators)

--keychain-uid <keychain_uid>

Overridden UID for asymmetric key

--sym-cipher-algo <sym_cipher_algo>

Optional intermediate symmetric cipher, to avoid stacking trustees

Options
AES_CBC | AES_EAX | CHACHA20_POLY1305

add-key-shard

Add a shard configuration to a shared secret

```
flightbox cryptoconf generate-simple add-key-shard [OPTIONS]
```

Options

--asym-cipher-algo <asym_cipher_algo>

Required Asymmetric algorithms for key encryption

Options
RSA_OAEP

--trustee-type <trustee_type>

Required Kind of key-guardian used

Options
local_keyfactory | authenticator

--keystore-uid <keystore_uid>

UID of the key-guardian (only for authenticators)

--keychain-uid <keychain_uid>

Overridden UID for asymmetric key

--sym-cipher-algo <sym_cipher_algo>

Optional intermediate symmetric cipher, to avoid stacking trustees

Options
AES_CBC | AES_EAX | CHACHA20_POLY1305

add-key-shared-secret

Transform a key into a shared secret

```
flightbox cryptoconf generate-simple add-key-shared-secret
    [OPTIONS]
```

Options

--threshold <threshold>

Required Number of key-guardians required for decryption of the secret

add-payload-cipher-layer

Add a layer of symmetric encryption of the data

The random symmetric key used for that encryption will then have to be protected by asymmetric encryption.

```
flightbox cryptoconf generate-simple add-payload-cipher-layer
    [OPTIONS]
```

Options

--sym-cipher-algo <sym_cipher_algo>

Required Symmetric algorithms for payload encryption

Options
AES_CBC | AES_EAX | CHACHA20_POLY1305

summarize

Display a summary of a cryptoconf structure

```
flightbox cryptoconf summarize [OPTIONS] CRYPTOCONF_FILE
```

Arguments

CRYPTOCONF_FILE

Required argument

validate

Ensure that a cryptoconf structure is valid

```
flightbox cryptoconf validate [OPTIONS] CRYPTOCONF_FILE
```

Arguments

CRYPTOCONF_FILE

Required argument

6.1.4 encrypt

Turn a media file into a secure container

```
flightbox encrypt [OPTIONS] INPUT_FILE
```

Options

-o, --output-basename <output_basename>

Basename of the cryptainer storage output file

-c, --cryptoconf <cryptoconf>

Json crypotoconf file

--bundle

Combine cryptainer metadata and payload

Arguments

INPUT_FILE

Required argument

6.1.5 foreign-keystore

Manage locally imported keystores

```
flightbox foreign-keystore [OPTIONS] COMMAND [ARGS]...
```

delete

Delete a locally imported keystore

```
flightbox foreign-keystore delete [OPTIONS] KEYSTORE_UID
```

Arguments

KEYSTORE_UID
    Required argument

import

Import a remote keystore

```
flightbox foreign-keystore import [OPTIONS]
```

Options

--from-usb
    Fetch authenticators from plugged USB devices

--from-path <from_path>
    Fetch authenticator from folder path

--from-gateway <from_gateway>
    Fetch authenticator by uid from gateway

--include-private-keys
    Import private keys when available

list

List locally imported keystores

```
flightbox foreign-keystore list [OPTIONS]
```

Options

-f, --format <format>

Default
    plain

Options
    plain | json

view

View metadata and public keypair identifiers of an imported keystore

The presence and validity of private keys isn't checked.

```
flightbox foreign-keystore view [OPTIONS] KEYSTORE_UID
```

Options

-f, --format <format>

> **Default**
> `plain`
>
> **Options**
> plain | json

Arguments

KEYSTORE_UID

Required argument

CHAPTER

SEVEN

SELECTED ALGORITHMS AND FORMATS

Here are the ciphers, protocols, data formats currently included in the Wacryptolib. The system is design so that new ones can easily be integrated, e.g. post-quantic ciphers, more compact data formats, etc.

7.1 Symmetric ciphers

- AES with CBC (cipher block chaining) mode. This mode applies a XOR operation between each block of plaintext and the previous ciphertext block, before encrypting it. As a result, the entire validity of all preceding blocks is contained in the immediately previous ciphertext block. This cipher uses an initialization vector (IV) of a certain length. The ciphertext is not authenticated, so its modification wouldn't be immediately visible when decrypting.
- AES with EAX (encrypt-then-authenticate-then-translate) mode : it is an Authenticated Encryption with Associated Data (AEAD) algorithm designed to simultaneously provide both authentication and privacy of the message. The reference implementation uses AES in CTR mode for encryption, combined with AES OMAC for authentication. EAX is a two-pass scheme, which means that encryption and authentication are done in separate operations. This algorithm has several desirable attributes, notably:
  - provable security (dependent on the security of the underlying primitive cipher);
  - message expansion is minimal, being limited to the overhead of the tag length;
  - using CTR mode means the cipher needs to be implemented only for encryption, simplifying implementation;
  - the algorithm is "on-line", that means that it can process a stream of data, using constant memory, without knowing total data length in advance;
  - the algorithm can process static Associated Data (AD), like session parameters in a communication;
- ChaCha20 with Poly1305: while this algorithm is mainly used for encryption, its core is a pseudo-random number generator. The cipher text is obtained by XOR'ing the plain text with a pseudo-random stream. Provided you never use the same nonce with the same key twice, you can treat that stream as a one time pad. This makes it very simple: unlike block ciphers, you don't have to worry about padding, and decryption is the same XOR operation as encryption. The Poly1305 authenticator prevents the insertion of fake messages into the stream.

7.2 Asymmetric ciphers

- RSA (Rivest–Shamir–Adleman) is one of the first public-key cryptosystems and is widely used for secure data transmission. In such a cryptosystem, the encryption key is public and it is different from the decryption key which is kept secret (private). In RSA, this asymmetry is based on the practical difficulty of the factorization of the product of two large prime numbers, the "factoring problem". A user of RSA creates and then publishes a public key based on two large prime numbers, along with an auxiliary value. The prime numbers must be kept secret. Anyone can use the public key to encrypt a message, but with currently published methods, and if the public key is large enough, only someone with knowledge of the prime numbers can decode the message feasibly. We use OAEP (Optimal Asymmetric Encryption Padding) as padding scheme to strengthen RSA encryption.

7.3 Signature algorithms

Signature is used to guarantee integrity and non-repudiation. Digital signatures are based on public key cryptography: the party that signs a message holds the private key, the one that verifies the signature holds the public key.

The cryptolib actually signs a hash of the message data concatenated with a timestamp.

- PSS (Probabilistic Signature Scheme) : The algorithm used to sign a plaintext is almost the same as the one used to cipher a plaintext in RSA; except that the private key is used to sign, and the public key to check if the signature is authentic.
- DSS (Digital Signature Standard): This algorithm works in the framework of DSA public-key cryptosystems and is based on the algebraic properties of modular exponentiation, together with the discrete logarithm problem, which is considered to be computationally intractable. The algorithm uses a key pair consisting of a public key and a private key. The private key is used to generate a digital signature for a message, and such a signature can be verified by using the signer's corresponding public key. The digital signature provides message authentication (the receiver can verify the origin of the message), integrity (the receiver can verify that the message has not been modified since it was signed) and non-repudiation (the sender cannot falsely claim that they have not signed the message). Also works with ECC keys, based on elliptic curves.

7.4 Hashing functions

Cryptographic hash functions take arbitrary binary strings as input, and produce a random-like fixed-length output (called digest or hash value). It is practically infeasible to derive the original input data from the digest. In other words, the cryptographic hash function is one-way (pre-image resistance). Given the digest of one message, it is also practically infeasible to find another message (second pre-image) with the same digest (weak collision resistance).

`wacryptolib.utilities.`**`SUPPORTED_HASH_ALGOS = ['SHA256', 'SHA512', 'SHA3_256', 'SHA3_512']`**

Hash algorithms authorized for use with *hash_message()*

7.5 Storage formats

Serialization of cryptoconfs and cryptainers is done using *Pymongo's Extended Json* format (*bson* might one day be used as an alternative). This Json dialect indeed supports many more data types: bytes, UUIDs, datetimes...

Hint: to convert base64 UUIDs stored in containers to a more natural hexadecimal form, use a converter like https://base64.guru/converter/decode/hex

The encrypted payload of cryptainers can either be stored as base64 string inside the JSON cryptainer (inline mode), or in a raw ciphertext file nearby (offloaded mode).

Unless specified otherwise, UTF8 is assumed as the encoding of all text data (ex. for decryption reports).

7.6 Communication protocols

Communication with remote APIs is done using JSON-RPC, but still with Pymongo's Extended Json dialect.

Errors are reported using our custom StatusSlugs implementation, which facilitate transmitting hierachical exceptions in webservices.

In the future, other API types (ex. RESTful) could be implemented similarly, to facilitate integration with other languages and platfoms.

7.7 Notes on safety and performance

- Encryption tags/macs should be computed on ciphertexts (encrypted payloads), not plaintext (initial data). Attempting decryption on crafted payloads is indeed an important attack vector, so integrity checks should occur before decryption, thanks to proper tags/macs. We shall rely on modern ciphers with AEAD (for Authenticated Encryption with Associated Data) to have both *confidentiality* and *integrity* in the same process.
- Security resides in the cryptosystem as a whole, not in individual algorithms. So it's more important to ensure that each workflow step is immune to main attack vectors, than to relentlessly seek safer algorithms and longer keys.
- Algorithms used should be part of easily accessible headers, not embedded into layers of multi-encrypted data. It is indeed more important to review these selected algorithms and detect broken/obsolete ones, than to hide them from potential attackers to attempt "security through obscurity".
- Compression of content must occur BEFORE encryption, since ciphertexts naturally have much higher entropy than plaintext. In particular, media data can often achieve high compression ratio at the cost of some accuracy loss.

CHAPTER

EIGHT

API DOCUMENTATION

8.1 Cryptainer

This module provides utilities to write and read encrypted cryptainers, which themselves use encryption/signing keys from trustees.

8.1.1 Cryptainer object processing

`wacryptolib.cryptainer.`**`encrypt_payload_into_cryptainer`**(*payload*, ***, *cryptoconf*, *cryptainer_metadata*, *keystore_pool=None*)

Turn a raw payload into a secure cryptainer, which can only be decrypted with the agreement of the owner and third-party trustees.

Parameters

- **payload** (`Union[bytes, BinaryIO]`) -- bytestring of media (image, video, sound...) or readable file object (file immediately deleted then)
- **cryptoconf** (`dict`) -- tree of specific encryption settings
- **cryptainer_metadata** (`Optional[dict]`) -- dict of metadata describing the payload (remains unencrypted in cryptainer)
- **keystore_pool** (`Optional[KeystorePoolBase]`) -- optional key storage pool, might be required by cryptoconf

Return type

`dict`

Returns

dict of cryptainer

`wacryptolib.cryptainer.`**`decrypt_payload_from_cryptainer`**(*cryptainer*, ***, *keystore_pool=None*, *passphrase_mapper=None*, *verify_integrity_tags=True*, *gateway_urls=None*, *revelation_requestor_uid=None*)

Decrypt a cryptainer with the help of third-parties.

Parameters

- **cryptainer** (`dict`) -- the cryptainer tree, which holds all information about involved keys
- **keystore_pool** (`Optional[KeystorePoolBase]`) -- optional key storage pool

- **passphrase_mapper** (`Optional[dict]`) -- optional dict mapping trustee IDs to their lists of passphrases
- **verify_integrity_tags** (`bool`) -- whether to check MAC tags of the ciphertext

Return type
`tuple`

Returns
tuple (data)

`wacryptolib.cryptainer.`**`extract_metadata_from_cryptainer`**(*cryptainer*)

Read the metadata tree (possibly None) from a cryptainer.

CURRENTLY CRYPTAINER METADATA ARE NEITHER ENCRYPTED NOR AUTHENTIFIED.

Parameters
cryptainer (`dict`) -- the cryptainer tree, which also holds cryptainer_metadata about encrypted content

Return type
`Optional[dict]`

Returns
dict

`wacryptolib.cryptainer.`**`get_cryptoconf_summary`**(*cryptoconf_or_cryptainer*)

Returns a string summary of the layers of encryption/signature of a cryptainer or a configuration tree.

`class` `wacryptolib.cryptainer.`**`CryptainerEncryptionPipeline`**(*cryptainer_filepath*, *, *cryptoconf*, *cryptainer_metadata*, *keystore_pool=None*, *dump_initial_cryptainer=True*)

Bases: `object`

Helper which prebuilds a cryptainer without signatures nor payload, fills its OFFLOADED ciphertext file chunk by chunk, and then dumps the final cryptainer (with signatures) to disk.

`wacryptolib.cryptainer.`**`encrypt_payload_and_stream_cryptainer_to_filesystem`**(*payload*, *, *cryptainer_filepath*, *cryptoconf*, *cryptainer_metadata*, *keystore_pool=None*)

Optimized version which directly streams encrypted payload to **offloaded** file, instead of creating a whole cryptainer and then dumping it to disk.

The cryptoconf used must be streamable with an EncryptionPipeline!

Return type
`None`

8.1.2 Validation utilities

`wacryptolib.cryptainer.`**`check_cryptainer_sanity`**`(`*cryptainer*, *jsonschema_mode=False*`)`

Validate the format of a cryptainer.

Parameters

`jsonschema_mode` -- If True, the cryptainer must have been loaded as raw json (with $binary, $numberInt and such) and will be checked using a jsonschema validator.

`wacryptolib.cryptainer.`**`check_cryptoconf_sanity`**`(`*cryptoconf*, *jsonschema_mode=False*`)`

Validate the format of a conf.

Parameters

`jsonschema_mode` -- If True, the cryptainer must have been loaded as raw json (with $binary, $numberInt and such) and will be checked using a jsonschema validator.

8.1.3 Filesystem operations

`wacryptolib.cryptainer.`**`dump_cryptainer_to_filesystem`**`(`*cryptainer_filepath*, *cryptainer*, *offload_payload_ciphertext=True*`)`

Dump a cryptainer to a file path, overwriting it if existing.

If *offload_payload_ciphertext*, actual encrypted payload is dumped to a separate bytes file nearby the json-formatted cryptainer.

Return type

`None`

`wacryptolib.cryptainer.`**`load_cryptainer_from_filesystem`**`(`*cryptainer_filepath*, *include_payload_ciphertext=True*`)`

Load a json-formatted cryptainer from a file path, potentially loading its offloaded ciphertext from a separate nearby bytes file.

Field *payload_ciphertext* is only present in result dict if *include_payload_ciphertext* is True.

Return type

`dict`

`wacryptolib.cryptainer.`**`delete_cryptainer_from_filesystem`**`(`*cryptainer_filepath*`)`

Delete a cryptainer file and its potential offloaded payload file.

`wacryptolib.cryptainer.`**`get_cryptainer_size_on_filesystem`**`(`*cryptainer_filepath*`)`

Return the total size in bytes occupied by a cryptainer and its potential offloaded payload file.

8.1.4 Cryptainer storage system

`class wacryptolib.cryptainer.`**`ReadonlyCryptainerStorage`**`(`*cryptainer_dir*, *keystore_pool=None*`)`

Bases: `object`

This class provides read access to a directory filled with cryptainers..

Parameters

- **`cryptainer_dir`** (`Path`) -- the folder where cryptainer files are stored
- **`keystore_pool`** (`Optional[KeystorePoolBase]`) -- optional KeystorePool, which might be required by current cryptoconf

check_cryptainer_sanity(*cryptainer_name_or_idx*)

Allows the validation of a cryptainer structure

decrypt_cryptainer_from_storage(*cryptainer_name_or_idx*, *passphrase_mapper=None*, *verify_integrity_tags=True*, *gateway_urls=None*, *revelation_requestor_uid=None*)

Return the decrypted content of the cryptainer *cryptainer_name_or_idx* (which must be in *list_cryptainer_names()*, or an index suitable for this sorted list).

Return type
`tuple`

list_cryptainer_names(*as_sorted_list=False*, *as_absolute_paths=False*, *finished=True*)

Returns the list of encrypted cryptainers present in storage, sorted by name or not, absolute or not, as Path objects.

If *finished* `is None, both finished and pending cryptainers are listed.

list_cryptainer_properties(*as_sorted_list=False*, *with_creation_datetime=False*, *with_age=False*, *with_size=False*, *with_offloaded=False*, *finished=True*)

Returns an list of dicts (unsorted by default) having the fields "name", [age] and [size], depending on requested properties.

load_cryptainer_from_storage(*cryptainer_name_or_idx*, *include_payload_ciphertext=True*)

Return the encrypted cryptainer dict for *cryptainer_name_or_idx* (which must be in *list_cryptainer_names()*, or an index suitable for this sorted list).

Only FINISHED cryptainers are expected to be loaded.

Return type
`dict`

class wacryptolib.cryptainer.**CryptainerStorage**(*cryptainer_dir*, *keystore_pool=None*, *default_cryptoconf=None*, *max_cryptainer_quota=None*, *max_cryptainer_count=None*, *max_cryptainer_age=None*, *max_workers=1*, *offload_payload_ciphertext=True*)

Bases: *ReadonlyCryptainerStorage*

This class encrypts file streams and stores them into filesystem, in a thread-safe way.

Exceeding cryptainers are automatically purged when enqueuing new files or waiting for idle state. A thread pool is used to encrypt files in the background.

Parameters

- **cryptainers_dir** -- the folder where cryptainer files are stored
- **keystore_pool** (`Optional[KeystorePoolBase]`) -- optional KeystorePool, which might be required by current cryptoconf
- **default_cryptoconf** (`Optional[dict]`) -- cryptoconf to use when none is provided when enqueuing payload
- **max_cryptainer_quota** (`Optional[int]`) -- if set, cryptainers are deleted if they exceed this size in bytes
- **max_cryptainer_count** (`Optional[int]`) -- if set, oldest exceeding cryptainers (time taken from their name, else their file-stats) are automatically erased

- **max_cryptainer_age** (`Optional[timedelta]`) -- if set, cryptainers exceeding this age (taken from their name, else their file-stats) in days are automatically erased
- **max_workers** (`int`) -- count of worker threads to use in parallel
- **offload_payload_ciphertext** -- whether actual encrypted payload must be kept separated from structured cryptainer file

create_cryptainer_encryption_stream(*filename_base*, *cryptainer_metadata*, *cryptoconf=None*, *dump_initial_cryptainer=True*, *cryptainer_encryption_stream_class=None*, *cryptainer_encryption_stream_extra_kwargs=None*)

Create and return a cryptainer encryption stream.

Purges exceeding cryptainers and pending results beforehand.

encrypt_file(*filename_base*, *payload*, *cryptainer_metadata*, *cryptoconf=None*)

Synchronously encrypt the provided payload into cryptainer storage.

Does NOT purge exceeding cryptainers and pending results beforehand.

Returns the cryptainer basename.

Return type

`str`

enqueue_file_for_encryption(*filename_base*, *payload*, *cryptainer_metadata*, *cryptoconf=None*)

Enqueue a payload for asynchronous encryption and storage.

Purges exceeding cryptainers and pending results beforehand.

The filename of final cryptainer might be different from provided one. Deware, target cryptainer with the same constructed name might be overwritten.

Parameters

- **payload** -- Bytes string, or a file-like object open for reading, which will be automatically closed.
- **cryptainer_metadata** -- Dict of metadata added (unencrypted) to cryptainer.
- **keychain_uid** -- If provided, replaces autogenerated default keychain_uid for this cryptainer.
- **cryptoconf** -- If provided, replaces default cryptoconf for this cryptainer.

wait_for_idle_state()

Wait for each pending future to be completed.

8.1.5 Trustee operations

`wacryptolib.cryptainer.`**get_trustee_proxy**(*trustee*, *keystore_pool*)

Return an TrusteeApi subclass instance (or proxy) depending on the content of *trustee* dict.

`wacryptolib.cryptainer.`**gather_trustee_dependencies**(*cryptainers*)

Analyse a cryptainer and return the trustees (and their keypairs) used by it.

Return type

`dict`

Returns

dict with lists of keypair identifiers in fields "encryption" and "signature".

`wacryptolib.cryptainer.`**`request_decryption_authorizations`**`(`*trustee_dependencies*, *keystore_pool*, *request_message*, *passphrases=None*`)`

Loop on encryption trustees and request decryption authorization for all the keypairs that they own.

Return type
`dict`

Returns
dict mapping trustee ids to authorization result dicts.

8.2 Trustee

This module provides base classes and utilities for trustee actors.

8.2.1 API for trustee services

`class wacryptolib.trustee.`**`TrusteeApi`**`(`*keystore*`)`

Bases: `object`

This is the API meant to be exposed by trustee webservices, to allow end users to create safely encrypted cryptainers.

Subclasses must add their own permission checking, especially so that no decryption with private keys can occur outside the scope of a well defined legal procedure.

`decrypt_with_private_key``(`*, *keychain_uid*, *cipher_algo*, *cipherdict*, *passphrases=None*, *cryptainer_metadata=None*`)`

Return the message (probably a symmetric key) decrypted with the corresponding key, as bytestring. Here again passphrases and cryptainer_metadata can be provided.

Raises if key existence, authorization or passphrase errors occur.

Return type
`bytes`

`fetch_public_key``(`*, *keychain_uid*, *key_algo*, *must_exist=False*`)`

Return a public key in PEM format bytestring, that caller shall use to encrypt its own symmetric keys, or to check a signature.

If *must_exist* is True, key is not autogenerated, and a KeyDoesNotExist might be raised.

Return type
`bytes`

`get_message_signature``(`*, *message*, *keychain_uid*, *signature_algo*`)`

Return a signature structure corresponding to the provided key and signature types.

Return type
`dict`

`request_decryption_authorization``(`*keypair_identifiers*, *request_message*, *passphrases=None*, *cryptainer_metadata=None*`)`

Send a list of keypairs for which decryption access is requested, with the reason why.

If request is immediately denied, an exception is raised, else the status of the authorization process (process which might involve several steps, including live encounters) is returned.

Parameters

- **keypair_identifiers** (Sequence) -- list of dicts with (keychain_uid, key_algo) indices to authorize
- **request_message** (str) -- user text explaining the reasons for the decryption (and the legal procedures involved)
- **passphrases** (Optional[Sequence]) -- optional list of passphrases to be tried on private keys
- **cryptainer_metadata** (Optional[dict]) -- metadata of the concerned cryptainer

Return type
`dict`

Returns
a dict with at least a string field "response_message" detailing the status of the request.

`class wacryptolib.trustee.ReadonlyTrusteeApi(`*keystore*`)`

Bases: *TrusteeApi*

Alternative Trustee API which relies on a fixed set of keys (e.g. imported from a key-device).

This version never generates keys by itself, whatever the values of method parameters like *must_exist*.

8.3 Authenticator

This module initializes and loads "authenticators", i.e sets of protected keys stored in a folder and belonging to a Key Guardian.

`wacryptolib.authenticator.initialize_authenticator(`*authenticator_dir*, *keystore_owner*, *keystore_passphrase_hint*`)`

BEWARE - PRIVATE API FOR NOW

Initialize a specific folder by creating a metadata file in it.

The folder must not be already initialized. It may not exist yet, but its parents must exist.

Parameters

- **authenticator_dir** (Path) -- Folder where the metadata file is expected.
- **keystore_owner** (str) -- owner name to store in device.
- **keystore_passphrase_hint** (str) -- hint for the passphrase used on private keys.

Return type
`dict`

Returns
(dict) Metadata for this authenticator.

`wacryptolib.authenticator.is_authenticator_initialized(`*authenticator_dir*`)`

BEWARE - PRIVATE API FOR NOW

Check if an authenticator folder SEEMS initialized.

Doesn't actually load the authenticator metadata file, nor check related keypairs.

Parameters
authenticator_dir (Path) -- (Path) folder where the metadata file is expected.

Returns
(bool) True if and only if the authenticator seems initialized.

8.4 Authentication device

This module detects potential "authentication devices", typically USB keys which can store the owner's authenticator.

`wacryptolib.authdevice.`**`list_available_authdevices`**`()`

Generate a list of dictionaries representing mounted partitions of USB keys.

Return type
`list`

Returns

list of dicts having at least these fields:

- "device_type" (str): device type like "USBSTOR"
- "partition_label" (str): possibly empty, label of the partition
- "partition_mountpoint" (str): mount point of device on the filesystem.
- "filesystem_format" (str): lowercase character string for filesystem type, like "ext2", "fat32" ...
- "filesystem_size" (int): filesystem size in bytes
- "authenticator_dir" (Path): Theoretical absolute path to the authenticator (might not exist yet)

8.5 Key generation

This module is dedicated to key generation, especially asymmetric public/private key pairs.

Note that keys are separated by use, thus keys of type RSA_OAEP (encryption) and RSA_PSS (signature) are different even for the same keychain uid.

8.5.1 Public API

`wacryptolib.keygen.`**`SUPPORTED_ASYMMETRIC_KEY_ALGOS = ['DSA_DSS', 'ECC_DSS', 'RSA_OAEP', 'RSA_PSS']`**

These values can be used as 'key_algo' for asymmetric key generation.

`wacryptolib.keygen.`**`generate_keypair`**`(*, *key_algo*, *serialize=True*, *key_length_bits=2048*, *curve='p521'*, *passphrase=None*)`

Generate a (public_key, private_key) pair.

Parameters

- **`key_algo`** (`str`) -- name of the key type
- **`serialize`** -- indicates if key must be serialized as PEM string (else it remains a python object)
- **`passphrase`** (`Optional[AnyStr]`) -- bytestring used for private key export (requires serialize=True)

Other arguments are used or not depending on the chosen *key_algo*.

Return type
`dict`

Returns
dictionary with "private_key" and "public_key" fields as objects or PEM-format strings

`wacryptolib.keygen.`**`load_asymmetric_key_from_pem_bytestring`**(*key_pem*, ***, *key_algo*, *passphrase=None*)

Load a key (public or private) from a PEM-formatted bytestring.

Parameters

- **`key_pem`** (`bytes`) -- the key bytrestring
- **`key_algo`** (`str`) -- name of the key format

Returns
key object

`wacryptolib.keygen.`**`SUPPORTED_SYMMETRIC_KEY_ALGOS = ['AES_CBC', 'AES_EAX', 'CHACHA20_POLY1305']`**

These values can be used as 'key_algo' for symmetric key generation.

`wacryptolib.keygen.`**`generate_symkey`**(*cipher_algo*)

Generate the strongest dict of keys/initializers possible for the wanted symmetric cipher, as a dict.

Return type
`dict`

8.5.2 Private API

The functions below are only documented for the details they give on specific arguments.

RSA

`wacryptolib.keygen.`**`_generate_rsa_keypair_as_objects`**(*key_length_bits*)

Generate a RSA (public_key, private_key) pair.

Parameters
`key_length_bits` (`int`) -- length of the key in bits, must be superior to 2048.

Return type
`dict`

Returns
dictionary with "private_key" and "public_key" fields as objects.

DSA

wacryptolib.keygen.**_generate_dsa_keypair_as_objects**(*key_length_bits*)

Generate a DSA (public_key, private_key) pair.

DSA keypair is not used for encryption/decryption, only for signing.

Parameters
key_length_bits (int) -- length of the key in bits, must be superior to 2048.

Return type
dict

Returns
dictionary with "private_key" and "public_key" fields as objects.

ECC

wacryptolib.keygen.**_generate_ecc_keypair_as_objects**(*curve*)

Generate an ECC (public_key, private_key) pair.

ECC keypair is not used for encryption/decryption, only for signing.

Parameters
curve (str) -- curve chosen among p256, p384, p521 and maybe others.

Return type
dict

Returns
dictionary with "private_key" and "public_key" fields as objects.

8.6 Key storage

This module provides classes for the storage of asymmetric key pairs.

8.6.1 Keystore

Each of these key storages theoretically belongs to a single user.

wacryptolib.keystore.**load_keystore_metadata**(*keystore_dir*)

Return the authenticator metadata dict stored in the given folder, after validating its format.

Raises KeystoreMetadataDoesNotExist if no metadata file is present.

Raises SchemaValidationError if keystore appears initialized, but has corrupted metadata.

Return type
dict

class wacryptolib.keystore.**InMemoryKeystore**

Bases: KeystoreReadWriteBase

Dummy key storage for use in tests, where keys are kepts only process-locally.

NOT MEANT TO BE THREAD-SAFE

`add_free_keypair`(**, key_algo, public_key, private_key*)

Store a pair of asymmetric keys into storage, free for subsequent attachment to an UUID.

Parameters

- **key_algo** (`str`) -- one of SUPPORTED_ASYMMETRIC_KEY_ALGOS
- **public_key** (`bytes`) -- public key in clear PEM format
- **private_key** (`bytes`) -- private key in PEM format (potentially encrypted)

Return type

`None`

`attach_free_keypair_to_uuid`(**, keychain_uid, key_algo*)

Fetch one of the free keypairs of storage of type *key_algo*, and attach it to UUID *keychain_uid*.

If no free keypair is available, a KeyDoesNotExist is raised.

Parameters

- **keychain_uid** (`UUID`) -- unique ID of the keychain
- **key_algo** (`str`) -- one of SUPPORTED_ASYMMETRIC_KEY_ALGOS

Return type

`None`

Returns

public key of the keypair, in clear PEM format

`get_free_keypairs_count`(*key_algo*)

Calculate the count of keypairs of type *key_algo* which are free for subsequent attachment to an UUID.

Parameters

key_algo (`str`) -- one of SUPPORTED_ASYMMETRIC_KEY_ALGOS

Return type

`int`

Returns

count of free keypairs of said type

`get_private_key`(**, keychain_uid, key_algo*)

Fetch a private key from persistent storage.

Parameters

- **keychain_uid** (`UUID`) -- unique ID of the keychain
- **key_algo** (`str`) -- one of SUPPORTED_ASYMMETRIC_KEY_ALGOS

Return type

`bytes`

Returns

private key in PEM format (potentially passphrase-protected), or raise KeyDoesNotExist

`get_public_key`(**, keychain_uid, key_algo*)

Fetch a public key from persistent storage.

Parameters

- **keychain_uid** (`UUID`) -- unique ID of the keychain
- **key_algo** (`str`) -- one of SUPPORTED_ASYMMETRIC_KEY_ALGOS

Return type
`bytes`

Returns
public key in clear PEM format, or raise KeyDoesNotExist

`list_keypair_identifiers()`

List identifiers of PUBLIC keys present in the storage, along with their potential private key existence.

Might raise an OperationNotSupported exception if not supported by this keystore.

Return type
`list`

Returns
a SORTED list of key information dicts with standard fields "keychain_uid" and "key_algo", as well as a boolean "private_key_present" which is True if the related private key exists in storage. Sorting is done by keychain_uid and then key_algo.

`set_keypair`(*, *keychain_uid*, *key_algo*, *public_key*, *private_key*)

Store a pair of asymmetric keys into storage, attached to a specific UUID.

Must raise a KeyAlreadyExists exception if a public/private key already exists for these uid/type identifiers.

Parameters

- **keychain_uid** (`UUID`) -- unique ID of the keychain
- **key_algo** (`str`) -- one of SUPPORTED_ASYMMETRIC_KEY_ALGOS
- **public_key** (`bytes`) -- public key in clear PEM format
- **private_key** (`bytes`) -- private key in PEM format (potentially encrypted)

Return type
`None`

`set_private_key`(*, *keychain_uid*, *key_algo*, *private_key*)

Store a private key, which must not already exist - else a KeyAlreadyExists is raised.

Important : the PUBLIC key for this private key must already exist in the keystore, else KeyDoesNotExist is raised.

Parameters

- **keychain_uid** -- unique ID of the keychain
- **key_algo** -- one of SUPPORTED_ASYMMETRIC_KEY_ALGOS
- **private_key** (`bytes`) -- private key in PEM format (potentially encrypted)

Return type
`None`

`set_public_key`(*, *keychain_uid*, *key_algo*, *public_key*)

Store a public key, which must not already exist - else KeyAlreadyExists is raised.

Parameters

- **keychain_uid** -- unique ID of the keychain
- **key_algo** -- one of SUPPORTED_ASYMMETRIC_KEY_ALGOS
- **public_key** (`bytes`) -- public key in clear PEM format

Return type
`None`

class `wacryptolib.keystore.FilesystemKeystore`(*keys_dir*)

Bases: `ReadonlyFilesystemKeystore`, `KeystoreReadWriteBase`

Filesystem-based key storage.

Protected by a process-wide lock, but not safe to use in multiprocessing environment, or in a process which can be brutally shutdown. To prevent corruption, caller should only persist UUIDs when the key storage operation is successfully finished.

`add_free_keypair`(**, key_algo, public_key, private_key*)

Store a pair of asymmetric keys into storage, free for subsequent attachment to an UUID.

Parameters

- **key_algo** (`str`) -- one of SUPPORTED_ASYMMETRIC_KEY_ALGOS
- **public_key** (`bytes`) -- public key in clear PEM format
- **private_key** (`bytes`) -- private key in PEM format (potentially encrypted)

Return type
`None`

`attach_free_keypair_to_uuid`(**, keychain_uid, key_algo*)

Fetch one of the free keypairs of storage of type *key_algo*, and attach it to UUID *keychain_uid*.

If no free keypair is available, a KeyDoesNotExist is raised.

Parameters

- **keychain_uid** (`UUID`) -- unique ID of the keychain
- **key_algo** (`str`) -- one of SUPPORTED_ASYMMETRIC_KEY_ALGOS

Return type
`None`

Returns
public key of the keypair, in clear PEM format

`export_to_keystore_tree`(*include_private_keys=True*)

Export keystore metadata and keys (public and, if include_private_keys is true, private) to a data tree.

`get_free_keypairs_count`(*key_algo*)

Calculate the count of keypairs of type *key_algo* which are free for subsequent attachment to an UUID.

Parameters
key_algo (`str`) -- one of SUPPORTED_ASYMMETRIC_KEY_ALGOS

Return type
`int`

Returns
count of free keypairs of said type

`get_keystore_metadata`(*include_keypair_identifiers=False*)

Return a metadata dict for the filesystem keystore, or raise KeystoreMetadataDoesNotExist.

`get_private_key`(**, keychain_uid, key_algo*)

Fetch a private key from persistent storage.

Parameters

- **keychain_uid** (`UUID`) -- unique ID of the keychain
- **key_algo** (`str`) -- one of SUPPORTED_ASYMMETRIC_KEY_ALGOS

Return type
`bytes`

Returns
private key in PEM format (potentially passphrase-protected), or raise KeyDoesNotExist

`get_public_key`(**, keychain_uid, key_algo*)

Fetch a public key from persistent storage.

Parameters

- **keychain_uid** (`UUID`) -- unique ID of the keychain
- **key_algo** (`str`) -- one of SUPPORTED_ASYMMETRIC_KEY_ALGOS

Return type
`bytes`

Returns
public key in clear PEM format, or raise KeyDoesNotExist

`import_from_keystore_tree`(*keystore_tree*)

Import keystore metadata and keys (public and, if included, private) fom a data tree.

If keystore already exists, it is completed with new keys, but metadata are untouched.

Returns True if and only if keystore was updated instead of created.

Return type
`bool`

`list_keypair_identifiers`()

List identifiers of PUBLIC keys present in the storage, along with their potential private key existence.

Might raise an OperationNotSupported exception if not supported by this keystore.

Return type
`list`

Returns
a SORTED list of key information dicts with standard fields "keychain_uid" and "key_algo", as well as a boolean "private_key_present" which is True if the related private key exists in storage. Sorting is done by keychain_uid and then key_algo.

`set_keypair`(**, keychain_uid, key_algo, public_key, private_key*)

Store a pair of asymmetric keys into storage, attached to a specific UUID.

Must raise a KeyAlreadyExists exception if a public/private key already exists for these uid/type identifiers.

Parameters

- **keychain_uid** (`UUID`) -- unique ID of the keychain
- **key_algo** (`str`) -- one of SUPPORTED_ASYMMETRIC_KEY_ALGOS
- **public_key** (`bytes`) -- public key in clear PEM format

- **`private_key`** (`bytes`) -- private key in PEM format (potentially encrypted)

Return type
`None`

`set_private_key`(***, *keychain_uid*, *key_algo*, *private_key*)

Store a private key, which must not already exist - else a KeyAlreadyExists is raised.

Important : the PUBLIC key for this private key must already exist in the keystore, else KeyDoesNotExist is raised.

Parameters

- **`keychain_uid`** -- unique ID of the keychain
- **`key_algo`** -- one of SUPPORTED_ASYMMETRIC_KEY_ALGOS
- **`private_key`** (`bytes`) -- private key in PEM format (potentially encrypted)

Return type
`None`

`set_public_key`(***, *keychain_uid*, *key_algo*, *public_key*)

Store a public key, which must not already exist - else KeyAlreadyExists is raised.

Parameters

- **`keychain_uid`** -- unique ID of the keychain
- **`key_algo`** -- one of SUPPORTED_ASYMMETRIC_KEY_ALGOS
- **`public_key`** (`bytes`) -- public key in clear PEM format

Return type
`None`

8.6.2 Keystore pools

These combine local and imported key storages under a single interface.

`class wacryptolib.keystore.`**`InMemoryKeystorePool`**

Bases: `KeystorePoolBase`

Dummy key storage pool for use in tests, where keys are kepts only process-locally.

NOT MEANT TO BE THREAD-SAFE

`class wacryptolib.keystore.`**`FilesystemKeystorePool`**(*root_dir*)

Bases: `KeystorePoolBase`

This class handles a set of locally stored key storages.

The local storage represents the current device/owner, and is expected to be used by read-write trustees, whereas imported key storages are supposed to be readonly, and only filled with keypairs imported from key-devices.

`export_foreign_keystore_to_keystore_tree`(*keystore_uid*, *include_private_keys=True*)

Exports data tree from the keystore targeted by keystore_uid.

`get_all_foreign_keystore_metadata`(*include_keypair_identifiers=False*)

Return a dict mapping key storage UUIDs to the dicts of their metadata.

Raises if any metadata loading fails.

Return type
`dict`

`get_foreign_keystore`(*keystore_uid*, *writable=False*)
The selected storage MUST exist, else a KeystoreDoesNotExist is raised.

`get_foreign_keystore_metadata`(*keystore_uid*, *include_keypair_identifiers=False*)
Return a metadata dict for the keystore *keystore_uid*.

`get_local_keyfactory`()
Storage automatically created if unexisting.

`import_foreign_keystore_from_keystore_tree`(*keystore_tree*)
Imports/updates data tree into the keystore targeted by keystore_uid.

Return type
`bool`

`list_foreign_keystore_uids`()
Return a sorted list of UUIDs of key storages, corresponding to the keystore_uid of their origin authentication devices.

Return type
`list`

8.7 Signature

This module allows to sign messages, and then to verify the signature.

`wacryptolib.signature.SUPPORTED_SIGNATURE_ALGOS = ['DSA_DSS', 'ECC_DSS', 'RSA_PSS']`
These values can be used as 'payload_signature_algo' parameters.

`wacryptolib.signature.sign_message`(*message*, ***, *signature_algo*, *private_key*)
Return a timestamped signature of the chosen type for the given payload, with the provided key (which must be of a compatible type).

Signature is actually performed on a SHA512 DIGEST of the message.

Parameters

- **message** (`bytes`) -- the bytestring to sign
- **signature_algo** (`str`) -- the name of the signing algorithm
- **private_key** (`object`) -- the cryptographic key used to create the signature

Return type
`dict`

Returns
dictionary with signature data

`wacryptolib.signature.verify_message_signature`(***, *message*, *signature_algo*, *signature*, *public_key*)
Verify the authenticity of a signature.

Raises if signature is invalid.

Parameters

- **message** (`bytes`) -- the bytestring which was signed

- **signature_algo** (str) -- the name of the signing algorithm
- **signature** (dict) -- structure describing the signature
- **public_key** (object) -- the cryptographic key used to verify the signature

8.8 Encryption

This module allows to encrypt bytestring data, then decrypt it.

8.8.1 Public API

wacryptolib.cipher.**SUPPORTED_CIPHER_ALGOS = ['AES_CBC', 'AES_EAX', 'CHACHA20_POLY1305', 'RSA_OAEP']**

These values can be used as 'cipher_algo'.

wacryptolib.cipher.**AUTHENTICATED_CIPHER_ALGOS = ['AES_EAX', 'CHACHA20_POLY1305']**

Built-in mutable sequence.

If no argument is given, the constructor creates a new empty list. The argument must be an iterable if specified.

wacryptolib.cipher.**STREAMABLE_CIPHER_ALGOS = ['AES_CBC', 'AES_EAX', 'CHACHA20_POLY1305']**

Built-in mutable sequence.

If no argument is given, the constructor creates a new empty list. The argument must be an iterable if specified.

wacryptolib.cipher.**encrypt_bytestring**(*plaintext*, *, *cipher_algo*, *key_dict*)

Encrypt a bytestring with the selected algorithm for the given payload, using the provided key dict (which must contain keys/initializers of proper types and lengths).

Return type
dict

Returns
dictionary with encryption data

wacryptolib.cipher.**decrypt_bytestring**(*cipherdict*, *, *cipher_algo*, *key_dict*, *verify_integrity_tags=True*)

Decrypt a bytestring with the selected algorithm for the given encrypted data dict, using the provided key (which must be of a compatible type and length).

Parameters

- **cipherdict** (dict) -- dict with field "ciphertext" as bytestring and (depending on the cipher_algo) some other fields like "tag" or "nonce" as bytestrings
- **cipher_algo** (str) -- one of the supported encryption algorithms
- **key_dict** (dict) -- dict with secret key fields
- **verify_integrity_tags** (bool) -- whether to check MAC tags of the ciphertext

Return type
bytes

Returns
dictionary with encryption data.

class `wacryptolib.cipher.`**`PayloadEncryptionPipeline`**(*output_stream*, *payload_cipher_layer_extracts*)

Bases: `object`

PRIVATE API FOR NOW

Pipeline to encrypt data through several encryption nodes, and stream it to an output binary stream (e.g. file or ByteIO)

8.8.2 Private API

The objects below are only documented for the details they give on specific arguments.

AES with CBC mode

class `wacryptolib.cipher.`**`AesCbcEncryptionNode`**(*key_dict*, *payload_digest_algo=()*)

Bases: `EncryptionNodeBase`

Encrypt a bytestring using AES (CBC mode).

`wacryptolib.cipher.`**`_encrypt_via_aes_cbc`**(*plaintext*, *key_dict*)

Encrypt a bytestring using AES (CBC mode).

Parameters

- **plaintext** (`bytes`) -- the bytes to cipher
- **key_dict** (`dict`) -- dict with AES cryptographic main key and iv. Main key must be 16, 24 or 32 bytes long (respectively for *AES-128*, *AES-192* or *AES-256*).

Return type
`dict`

Returns
dict with field "ciphertext" as bytestring

`wacryptolib.cipher.`**`_decrypt_via_aes_cbc`**(*cipherdict*, *key_dict*, *verify_integrity_tags=True*)

Decrypt a bytestring using AES (CBC mode).

Parameters

- **cipherdict** (`dict`) -- dict with field "ciphertext" as bytestring
- **key_dict** (`dict`) -- dict with AES cryptographic main key and nonce.
- **verify_integrity_tags** (`bool`) -- whether to check MAC tags of the ciphertext (not applicable for this cipher)

Return type
`bytes`

Returns
the decrypted bytestring

AES with EAX mode

class wacryptolib.cipher.**AesEaxEncryptionNode**(*key_dict*, *payload_digest_algo=()*)

Bases: `EncryptionNodeBase`

Encrypt a bytestring using AES (EAX mode).

wacryptolib.cipher.**_encrypt_via_aes_eax**(*plaintext*, *key_dict*)

Encrypt a bytestring using AES (EAX mode).

Parameters

- **plaintext** (`bytes`) -- the bytes to cipher
- **key_dict** (`dict`) -- dict with AES cryptographic main key and nonce. Main key must be 16, 24 or 32 bytes long (respectively for *AES-128*, *AES-192* or *AES-256*).

Return type

`dict`

Returns

dict with fields "ciphertext" and "tag" as bytestrings

wacryptolib.cipher.**_decrypt_via_aes_eax**(*cipherdict*, *key_dict*, *verify_integrity_tags=True*)

Decrypt a bytestring using AES (EAX mode).

Parameters

- **cipherdict** (`dict`) -- dict with fields "ciphertext", "tag" as bytestrings
- **key_dict** (`dict`) -- dict with AES cryptographic main key and nonce.
- **verify_integrity_tags** (`bool`) -- whether to check MAC tags of the ciphertext

Return type

`bytes`

Returns

the decrypted bytestring

ChaCha20_Poly1305

class wacryptolib.cipher.**Chacha20Poly1305EncryptionNode**(*key_dict*, *payload_digest_algo=()*)

Bases: `EncryptionNodeBase`

Encrypt a bytestring using ChaCha20 with Poly1305 authentication.

wacryptolib.cipher.**_encrypt_via_chacha20_poly1305**(*plaintext*, *key_dict*)

Encrypt a bytestring with the stream cipher ChaCha20.

Additional cleartext data can be provided so that the generated mac tag also verifies its integrity.

Parameters

- **plaintext** (`bytes`) -- the bytes to cipher
- **key_dict** (`dict`) -- 32 bytes long cryptographic key and nonce

Return type

`dict`

Returns

dict with fields "ciphertext", "tag", and "header" as bytestrings `wacryptolib.cipher.`**`_decrypt_via_chacha20_poly1305`**(*cipherdict*, *key_dict*, *verify_integrity_tags=True*)

Decrypt a bytestring with the stream cipher ChaCha20.

Parameters

- **`cipherdict`** (`dict`) -- dict with fields "ciphertext", "tag" and "nonce" as bytestrings
- **`key_dict`** (`dict`) -- 32 bytes long cryptographic key and nonce
- **`verify_integrity_tags`** (`bool`) -- whether to check MAC tags of the ciphertext

Return type

`bytes`

Returns

the decrypted bytestring

RSA - PKCS#1 OAEP

`wacryptolib.cipher.`**`_encrypt_via_rsa_oaep`**(*plaintext*, *key_dict*)

Encrypt a bytestring with PKCS#1 RSA OAEP (asymmetric algo).

Parameters

- **`plaintext`** (`bytes`) -- the bytes to cipher
- **`key_dict`** (`dict`) -- dict with PUBLIC RSA key object (RSA.RsaKey)

Return type

`dict`

Returns

a dict with field *digest_list*, containing bytestring chunks of variable width.

`wacryptolib.cipher.`**`_decrypt_via_rsa_oaep`**(*cipherdict*, *key_dict*, *verify_integrity_tags=True*)

Decrypt a bytestring with PKCS#1 RSA OAEP (asymmetric algo).

Parameters

- **`cipherdict`** (`dict`) -- list of ciphertext chunks
- **`key_dict`** (`dict`) -- dict with PRIVATE RSA key object (RSA.RsaKey)
- **`verify_integrity_tags`** (`bool`) -- whether to check MAC tags of the ciphertext (not applicable for this cipher)

Return type

`bytes`

Returns

the decrypted bytestring

8.9 Shared secret

This module provides utilities to dissociate and recombine Shamir "shared secrets", for which some parts (shards) can be lost without preventing the reconstruction of the whole secret.

`wacryptolib.shared_secret.`**`split_secret_into_shards`**`(`*secret*, *, *shard_count*, *threshold_count*`)`

Generate a Shamir shared secret of *shard_count* subkeys, with *threshold_count* of them required to recompute the initial *bytestring*.

Parameters

- **`secret`** (`bytes`) -- bytestring to separate as shards, whatever its length
- **`shard_count`** (`int`) -- the number of shards to be created for the secret
- **`threshold_count`** (`int`) -- the minimal number of shards needed to recombine the key

Return type

`list`

Returns

list of full bytestring shards

`wacryptolib.shared_secret.`**`recombine_secret_from_shards`**`(`*shards*`)`

Reconstruct a secret from list of Shamir *shards*

Parameters

`shards` (`Sequence`) -- list of k full-length shards (k being exactly the threshold of this shared secret)

Return type

`bytes`

Returns

the key reconstructed as bytes

8.10 Sensor

This module provides base classes to create sensors (gps, gyroscope, audio, video...) and aggregate/push their data towards cryptainers.

8.10.1 Aggregation of records into binary archives

`class` `wacryptolib.sensor.`**`TarfileRecordAggregator`**`(`*cryptainer_storage*, *max_duration_s*`)`

Bases: `TimeLimitedAggregatorMixin`

This class allows sensors to aggregate file-like records of data in memory.

It is in charge of building the filenames of tar records, as well as of completed tarfiles.

Public methods of this class are thread-safe.

`add_record``(`*sensor_name*, *from_datetime*, *to_datetime*, *extension*, *payload*`)`

Add the provided data to the tarfile, using associated metadata.

If, despite included timestamps, several records end up having the exact same name, the last one will have priority when extracting the tarfile, but all of them will be stored in it anyway.

Parameters

- **`sensor_name`** (`str`) -- slug label for the sensor
- **`from_datetime`** (`datetime`) -- start time of the recording
- **`to_datetime`** (`datetime`) -- end time of the recording
- **`extension`** (`str`) -- file extension, starting with a dot
- **`payload`** (`bytes`) -- bytestring of audio/video/other data

`finalize_tarfile``()`

Return the content of current tarfile as a bytestring, possibly empty, and reset the current tarfile.

`static` **`read_tarfile_from_bytestring`**`(`*payload*`)`

Create a readonly TarFile instance from the provided bytestring.

8.10.2 Base classes for poller/pusher sensors

`class wacryptolib.sensor.`**`JsonDataAggregator`**`(`*tarfile_aggregator*, *sensor_name*, *max_duration_s*`)`

Bases: `TimeLimitedAggregatorMixin`

This class allows sensors to aggregate dicts of data, which are periodically pushed as a json bytestring to the underlying TarfileRecordAggregator.

Public methods of this class are thread-safe.

`add_data``(`*data_dict*`)`

Flush current data to the tarfile if needed, and append *data_dict* to the queue.

`flush_dataset``()`

Force the flushing of current data to the tarfile (e.g. when terminating the service).

`class wacryptolib.sensor.`**`PeriodicValuePoller`**`(`*json_aggregator*, ***kwargs*`)`

Bases: `PeriodicValueMixin`, *`PeriodicTaskHandler`*

This class runs a function at a specified interval, and pushes its result to a json aggregator.

8.10.3 Simultaneous management of multiple sensors

`class wacryptolib.sensor.`**`SensorManager`**`(`*sensors*`)`

Bases: *`TaskRunnerStateMachineBase`*

Manage a group of sensors for simultaneous starts/stops.

The underlying aggregators are not supposed to be directly impacted by these operations - they must be flushed separately.

`join``()`

Wait for the periodic system to really finish running. Does nothing if periodic system is already stopped.

`start``()`

Start the periodic system which will poll or push the value.

`stop``()`

Request the periodic system to stop as soon as possible.

8.11 Json-rpc client

This module provides a client for json-rpc webservices.

class `wacryptolib.jsonrpc_client.`**`JsonRpcProxy`**(*url*, *args, *response_error_handler=<function status_slugs_response_error_handler>*, **kwargs)

Bases: `Server`

A connection to a HTTP JSON-RPC server, backed by the *requests* library.

Parameters can be passed by position, or as keyword arguments (but not both).

See https://pypi.org/project/jsonrpc-requests/ for usage examples.

The differences between our *JsonRpcProxy* and upstream's *Server* class are:

- we dump/load data using Pymongo's Extended Json format, able to transparently deal with bytes, uuids, dates etc.
- we do not auto-unpack single dict arguments on call, e.g *proxy.foo({'fizz': 1, 'fuzz': 2})* will be treated as calling remote foo() with a single dict argument, not as passing it keyword arguments *fizz* and *fuzz*.
- a *response_error_handler* callback can be provided to swallow or convert an error received in an RPC response.

`wacryptolib.jsonrpc_client.`**`status_slugs_response_error_handler`**(*exc*)

Generic error handler which recognizes status slugs of builtin exceptions in json-rpc error responses, and reraises them client-side.

8.12 Utilities

This module exposes different functions which can be useful when dealing with cryptography and workers.

8.12.1 Task handling

`wacryptolib.utilities.`**`TaskRunnerStateMachineBase`**(***kwargs*)

State machine for all sensors/players, checking that the order of start/stop/join operations is correct.

The two-steps shutdown (*stop()*, and later *join()*) allows caller to efficiently and safely stop numerous runners.

`wacryptolib.utilities.`**`PeriodicTaskHandler`**(*interval_s*, *count=-1*, *runonstart=True*, *task_func=None*, ***kwargs*)

This class runs a task at a specified interval, with start/stop/join controls.

If *task_func* argument is not provided, then *_offloaded_run_task()* must be overridden by subclass.

8.12.2 Hashing

`wacryptolib.utilities.`**`SUPPORTED_HASH_ALGOS = ['SHA256', 'SHA512', 'SHA3_256', 'SHA3_512']`**

Hash algorithms authorized for use with *hash_message()*

`wacryptolib.utilities.`**`hash_message`**`(`*message*, *hash_algo*`)`

Hash a message with the selected hash algorithm, and return the hash as bytes.

8.12.3 Serialization

`wacryptolib.utilities.`**`dump_to_json_str`**`(`*data*, ***extra_options*`)`

Dump a data tree to a json representation as string. Supports advanced types like bytes, uuids, dates...

`wacryptolib.utilities.`**`load_from_json_str`**`(`*data*, ***extra_options*`)`

Load a data tree from a json representation as string. Supports advanced types like bytes, uuids, dates...

Raises exceptions.ValidationError on loading error.

`wacryptolib.utilities.`**`dump_to_json_bytes`**`(`*data*, ***extra_options*`)`

Same as *dump_to_json_str*, but returns UTF8-encoded bytes.

`wacryptolib.utilities.`**`load_from_json_bytes`**`(`*data*, ***extra_options*`)`

Same as *load_from_json_str*, but takes UTF8-encoded bytes as input.

`wacryptolib.utilities.`**`dump_to_json_file`**`(`*filepath*, *data*, ***extra_options*`)`

Same as *dump_to_json_bytes*, but writes data to filesystem (and returns bytes too).

`wacryptolib.utilities.`**`load_from_json_file`**`(`*filepath*, ***extra_options*`)`

Same as *load_from_json_bytes*, but reads data from filesystem.

8.12.4 Miscellaneous

`wacryptolib.utilities.`**`generate_uuid0`**`(`*ts=None*`)`

Generate a random UUID partly based on Unix timestamp (not part of official "variants").

Uses 6 bytes to encode the time and does not encode any version bits, leaving 10 bytes (80 bits) of random data.

When just transmitting these UUIDs around, the stdlib "uuid" module does the job fine, no need for uuid0 lib.

Parameters
`ts` (`Optional[float]`) -- optional timestamp to use instead of current time (if not falsey)

Returns
uuid0 object (subclass of UUID)

`wacryptolib.utilities.`**`split_as_chunks`**`(`*bytestring*, *, *chunk_size*, *must_pad*, *accept_incomplete_chunk=False*`)`

Split a *bytestring* into chunks (or blocks)

Parameters

- **`bytestring`** (`bytes`) -- element to be split into chunks
- **`chunk_size`** (`int`) -- size of a chunk in bytes
- **`must_pad`** (`bool`) -- whether the bytestring must be padded first or not

- **accept_incomplete_chunk** (bool) -- do not raise error if a chunk with a length != chunk_size is obtained

Return type
`List[bytes]`

Returns
list of bytes chunks

`wacryptolib.utilities.`**`recombine_chunks`**`(`*chunks*, *, *chunk_size*, *must_unpad*`)`

Recombine chunks which were previously separated.

Parameters

- **chunks** (`Sequence[bytes]`) -- sequence of bytestring parts
- **chunk_size** (`int`) -- size of a chunk in bytes (only used for error checking, when unpadding occurs)
- **must_unpad** (`bool`) -- whether the bytestring must be unpadded after recombining, or not

Return type
`bytes`

Returns
initial bytestring

8.13 Exceptions

These wacryptolib-specific exception classes should be used everywhere for functional errors, instead of standard Python exceptions (ValueError etc.)

exception `wacryptolib.exceptions.`**`FunctionalError`**

Bases: `Exception`

Base class for all 'normal' errors of the API

exception `wacryptolib.exceptions.`**`ExistenceError`**

Bases: *`FunctionalError`*

exception `wacryptolib.exceptions.`**`KeyDoesNotExist`**

Bases: *`ExistenceError`*

exception `wacryptolib.exceptions.`**`KeyAlreadyExists`**

Bases: *`ExistenceError`*

exception `wacryptolib.exceptions.`**`KeystoreDoesNotExist`**

Bases: *`ExistenceError`*

exception `wacryptolib.exceptions.`**`KeystoreAlreadyExists`**

Bases: *`ExistenceError`*

exception `wacryptolib.exceptions.`**`KeystoreMetadataDoesNotExist`**

Bases: *`KeystoreDoesNotExist`*

exception `wacryptolib.exceptions.`**`AuthenticationError`**

Bases: *`FunctionalError`*

exception wacryptolib.exceptions.**AuthorizationError**

Bases: *FunctionalError*

exception wacryptolib.exceptions.**OperationNotSupported**

Bases: *FunctionalError*

exception wacryptolib.exceptions.**CryptographyError**

Bases: *FunctionalError*

exception wacryptolib.exceptions.**EncryptionError**

Bases: *CryptographyError*

exception wacryptolib.exceptions.**DecryptionError**

Bases: *CryptographyError*

exception wacryptolib.exceptions.**DecryptionIntegrityError**

Bases: *DecryptionError*

exception wacryptolib.exceptions.**SignatureCreationError**

Bases: *CryptographyError*

exception wacryptolib.exceptions.**SignatureVerificationError**

Bases: *CryptographyError*

exception wacryptolib.exceptions.**KeyLoadingError**

Bases: *CryptographyError*

exception wacryptolib.exceptions.**ValidationError**

Bases: *FunctionalError*

exception wacryptolib.exceptions.**SchemaValidationError**

Bases: *ValidationError*

8.14 Error handling

This module provides utilities to convert python exceptions from/to a generic serialized format, so that webservice client can handle them in a hierarchical and forward-compatible manner.

class wacryptolib.error_handling.**StatusSlugMapper**(*exception_classes*, *fallback_exception_class*, *exception_slugifier=<function slugify_exception_class>*)

Bases: object

High-level wrapper for converting exceptions from/to status slugs.

static gather_exception_subclasses(*parent_classes*)

Browse the module's variables, and return all found exception classes which are subclasses of *parent_classes* (including these, if found in module).

Parameters

- **module** -- python module object
- **parent_classes** (Sequence) -- list of exception classes (or single exception class)

Returns

list of exception subclasses

`get_closest_exception_class_for_status_slugs`(*slugs*)

Return the closest exception class targeted by the provided status slugs, with a fallback class if no matching ancestor is found at all.

`slugify_exception_class`(*exception_class*, **args*, ***kwargs*)

Use the exception slugifier provided in *__init__()* to turn an exception class into a qualified name.

`wacryptolib.error_handling.gather_exception_subclasses`(*module*, *parent_classes*)

Browse the module's variables, and return all found exception classes which are subclasses of *parent_classes* (including these, if found in module).

Parameters

- `module` -- python module object
- `parent_classes` (`Sequence`) -- list of exception classes (or single exception class)

Returns

list of exception subclasses

`wacryptolib.error_handling.slugify_exception_class`(*exception_class*, *excluded_classes=(<class 'object'>, <class 'BaseException'>, <class 'Exception'>)*, *qualified_name_extractor=<function _fully_qualified_name>*)

Turn an exception class into a list of slugs which identifies it uniquely, from ancestor to descendant.

Parameters

- `exception_class` -- exception class to slugify
- `excluded_classes` -- list of parents classes so generic that they needn't be included in slugs
- `qualified_name_extractor` -- callable which turns an exception class into its qualified name

Returns

list of strings

`wacryptolib.error_handling.construct_status_slug_mapper`(*exception_classes*, *fallback_exception_class*, *exception_slugifier=<function slugify_exception_class>*)

Construct and return a tree where branches are qualified slugs, and each leaf is an exception class corresponding to the path leading to it.

Intermediate branches can carry an (ancestor) exception class too, but only if this one is explicitely included in *exception_classes*.

The fallback exception class is stored at the root of the tree under the "" key.

`wacryptolib.error_handling.get_closest_exception_class_for_status_slugs`(*slugs*, *mapper_tree*)

Return the exception class targeted by the provided status slugs, or the closest ancestor class if the exact exception class is not in the mapper.

If *slugs* is empty, or if no ancestor is found, the fallback exception of the mapper is returned instead.

Parameters

- `slugs` -- qualified status slugs
- `mapper_tree` -- mapper tree constructed from selected exceptions

Returns
exception class object

8.15 Scaffolding for tests

This module provides utilities so that other projects can easily test their own implementations of cryptolib-related classes.

This is a provisional API, which may change without deprecation period, and which requires dependencies like *pytest*.

`wacryptolib.scaffolding.`**`check_keystore_basic_get_set_api`**`(`*keystore*, *readonly_keystore=None*`)`
Test the workflow of getters/setters of the storage API, for uid-attached keys.

`wacryptolib.scaffolding.`**`check_keystore_free_keys_api`**`(`*keystore*`)`
Test the storage regarding the precreation of "free keys", and their subsequent attachment to uids.

`wacryptolib.scaffolding.`**`check_keystore_free_keys_concurrency`**`(`*keystore*`)`
Parallel tests to check the thread-safety of the storage regarding "free keys" booking.

`wacryptolib.scaffolding.`**`check_sensor_state_machine`**`(`*sensor*, *run_duration=0*`)`
Check the proper start/stop/join behaviour of a sensor instance.

CHAPTER

NINE

DEVELOPMENT INSTRUCTIONS

9.1 Getting started

To develop on the WACryptolib, the interpreter for *python3.7* or later must be installed (see *pyproject.toml* for version details).

Instead of pip, we use poetry to manage dependencies.

9.1.1 Automatic setup

Launch *python wacryptolib_installer.py* in repository root, from inside a python virtual environment.

This will update pip, install a *local* version of poetry, install the python modules required by wacryptolib, and then launch unit-tests.

On Windows, poetry might try to move some DLLs it is currently using, so install might crash with file permission errors, but relaunching the installer should eventually succeed...

9.1.2 Manual setup

Use *pip install poetry* to install poetry (or better, follow its official docs to install it system-wide and avoid the permission errors mentioned just above).

Use *poetry install* from repository root, to install python dependencies (poetry will create its own virtualenv if you don't have one activated).

As an alternative, you can launch *pip install -r pip_requirements_export.txt*, but this requirements file might be a bit outdated for latest Python versions.

9.1.3 Launching the CLI

To try the command line interface, the easiest is to launch the *main.py* script.

If you added "src/" to your pythonpath, e.g. with *pip install -e <repo-root>* (requires pip>=21.3), you can instead use:

```
$ python -m wacryptolib
```

When wacryptolib has been installed with pip, it exposes a **"flightbox"** executable which does the same as the main.py script.

9.1.4 Handy dev commands

Use *pytest* to launch unit-tests (default pytest arguments are in *setup.cfg*). Use *poetry run pytest* instead, if poetry manages its own virtualenv.

Add *--cov=wacryptolib* argument to the pytest command to generate coverage reports.

Use *bash ci.sh* to do a full checkup before committing or pushing your changes (under Windows, launch CI commands one by one).

Use the Black formatter to format the python code like so:

```
$ black -l 120 src/ tests/
```

To generate documentation, launch Sphinx commands ("make html"...) from the doc/subfolder. The "flightbox" entry-point mentioned above should have been installed into your virtualenv first, else, some documentation generation will fail.

9.2 Release process

To release a new version of the WACryptolib, we don't need Twine, since Poetry already has publishing commands.

9.2.1 Initial setup

You must first register testpypi as a valid package store in Poetry:

```
$ poetry config repositories.testpypi https://test.pypi.org/legacy/
```

Check it then with:

```
$ poetry config --list
```

9.2.2 Publish a new version

Issue:

```
$ poetry build
$ poetry publish -r testpypi
```

Then test this preview package in some project using the wacryptolib:

```
$ python -m pip install -U --index-url https://test.pypi.org/simple/ --extra-index-url␣
↪https://pypi.org/simple/ wacryptolib
```

When all is verified on testpypi (Readme, files uploaded, etc.), release the package to the real pypi:

```
$ poetry publish
```

If authentication troubles, try setting credentials with something like this:

```
$ poetry config http-basic.<testpypi/pypi> <username> <password>
```

CHAPTER TEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

W

`wacryptolib.exceptions`, 67

INDEX

Symbols

_decrypt_via_aes_cbc() (*in module wacryptolib.cipher*), 60
_decrypt_via_aes_eax() (*in module wacryptolib.cipher*), 61
_decrypt_via_chacha20_poly1305() (*in module wacryptolib.cipher*), 61
_decrypt_via_rsa_oaep() (*in module wacryptolib.cipher*), 62
_encrypt_via_aes_cbc() (*in module wacryptolib.cipher*), 60
_encrypt_via_aes_eax() (*in module wacryptolib.cipher*), 61
_encrypt_via_chacha20_poly1305() (*in module wacryptolib.cipher*), 61
_encrypt_via_rsa_oaep() (*in module wacryptolib.cipher*), 62
_generate_dsa_keypair_as_objects() (*in module wacryptolib.keygen*), 52
_generate_ecc_keypair_as_objects() (*in module wacryptolib.keygen*), 52
_generate_rsa_keypair_as_objects() (*in module wacryptolib.keygen*), 51
--asym-cipher-algo
  flightbox-cryptoconf-generate-simple-add-key-cipher-layer command line option, 34
  flightbox-cryptoconf-generate-simple-add-key-shard command line option, 34
--bundle
  flightbox-encrypt command line option, 36
--cryptainer-storage
  flightbox command line option, 29
--cryptoconf
  flightbox-encrypt command line option, 36
--format
  flightbox-authenticator-view command line option, 31
  flightbox-cryptainer-list command line option, 32
  flightbox-foreign-keystore-list command line option, 37
  flightbox-foreign-keystore-view command line option, 38
--from-gateway
  flightbox-foreign-keystore-import command line option, 37
--from-path
  flightbox-foreign-keystore-import command line option, 37
--from-usb
  flightbox-foreign-keystore-import command line option, 37
--gateway-url
  flightbox command line option, 29
--include-private-keys
  flightbox-foreign-keystore-import command line option, 37
--keychain-uid
  flightbox-cryptoconf-generate-simple command line option, 33
  flightbox-cryptoconf-generate-simple-add-key-cipher-la command line option, 34
  flightbox-cryptoconf-generate-simple-add-key-shard command line option, 35
--keypair-count
  flightbox-authenticator-create command line option, 30
--keystore-pool
  flightbox command line option, 29
--keystore-uid
  flightbox-cryptoconf-generate-simple-add-key-cipher-la command line option, 34
  flightbox-cryptoconf-generate-simple-add-key-shard command line option, 34
--max-age
  flightbox-cryptainer-purge command line option, 32
--max-count
  flightbox-cryptainer-purge command line option, 32
--max-quota
  flightbox-cryptainer-purge command line option, 32
--output-basename

flightbox-encrypt command line option, 36
--output-file
flightbox-cryptainer-decrypt command line option, 31
--owner
flightbox-authenticator-create command line option, 30
--passphrase-hint
flightbox-authenticator-create command line option, 30
--sym-cipher-algo
flightbox-cryptoconf-generate-simple-add-key-cipher-layer command line option, 34
flightbox-cryptoconf-generate-simple-add-key-shard command line option, 35
flightbox-cryptoconf-generate-simple-add-payload-cipher-layer command line option, 35
--threshold
flightbox-cryptoconf-generate-simple-add-key-shared-secret command line option, 35
--trustee-type
flightbox-cryptoconf-generate-simple-add-key-cipher-layer command line option, 34
flightbox-cryptoconf-generate-simple-add-key-shard command line option, 34
--verbosity
flightbox command line option, 29
-c
flightbox command line option, 29
flightbox-encrypt command line option, 36
-f
flightbox-authenticator-view command line option, 31
flightbox-cryptainer-list command line option, 32
flightbox-foreign-keystore-list command line option, 37
flightbox-foreign-keystore-view command line option, 38
-g
flightbox command line option, 29
-k
flightbox command line option, 29
-o
flightbox-cryptainer-decrypt command line option, 31
flightbox-encrypt command line option, 36
-v
flightbox command line option, 29

A

add_data() (*wacryptolib.sensor.JsonDataAggregator method*), 64
add_free_keypair() (*wacryptolib.keystore.FilesystemKeystore method*), 55
add_free_keypair() (*wacryptolib.keystore.InMemoryKeystore method*), 52
add_record() (*wacryptolib.sensor.TarfileRecordAggregator method*), 63
AesCbcEncryptionNode (*class in wacryptolib.cipher*), 60
AesEaxEncryptionNode (*class in wacryptolib.cipher*), 61
attach_free_keypair_to_uuid() (*wacryptolib.keystore.FilesystemKeystore method*), 55
attach_free_keypair_to_uuid() (*wacryptolib.keystore.InMemoryKeystore method*), 53
AUTHENTICATED_CIPHER_ALGOS (*in module wacryptolib.cipher*), 59
AuthenticationError, 67
AUTHENTICATOR_DIR
flightbox-authenticator-create command line option, 30
flightbox-authenticator-delete command line option, 30
flightbox-authenticator-validate command line option, 30
flightbox-authenticator-view command line option, 31
AuthorizationError, 67

C

Chacha20Poly1305EncryptionNode (*class in wacryptolib.cipher*), 61
check_cryptainer_sanity() (*in module wacryptolib.cryptainer*), 45
check_cryptainer_sanity() (*wacryptolib.cryptainer.ReadonlyCryptainerStorage method*), 45
check_cryptoconf_sanity() (*in module wacryptolib.cryptainer*), 45
check_keystore_basic_get_set_api() (*in module wacryptolib.scaffolding*), 70
check_keystore_free_keys_api() (*in module wacryptolib.scaffolding*), 70
check_keystore_free_keys_concurrency() (*in module wacryptolib.scaffolding*), 70
check_sensor_state_machine() (*in module wacryptolib.scaffolding*), 70
construct_status_slug_mapper() (*in module wacryptolib.error_handling*), 69
create_cryptainer_encryption_stream()

(*wacryptolib.cryptainer.CryptainerStorage method*), 47
CRYPTAINER_NAME
    flightbox-cryptainer-decrypt command line option, 32
    flightbox-cryptainer-delete command line option, 32
    flightbox-cryptainer-summarize command line option, 33
    flightbox-cryptainer-validate command line option, 33
CryptainerEncryptionPipeline (*class in wacryptolib.cryptainer*), 44
CryptainerStorage (*class in wacryptolib.cryptainer*), 46
CRYPTOCONF_FILE
    flightbox-cryptoconf-summarize command line option, 36
    flightbox-cryptoconf-validate command line option, 36
CryptographyError, 68

D

decrypt_bytestring() (*in module wacryptolib.cipher*), 59
decrypt_cryptainer_from_storage() (*wacryptolib.cryptainer.ReadonlyCryptainerStorage method*), 46
decrypt_payload_from_cryptainer() (*in module wacryptolib.cryptainer*), 43
decrypt_with_private_key() (*wacryptolib.trustee.TrusteeApi method*), 48
DecryptionError, 68
DecryptionIntegrityError, 68
delete_cryptainer_from_filesystem() (*in module wacryptolib.cryptainer*), 45
dump_cryptainer_to_filesystem() (*in module wacryptolib.cryptainer*), 45
dump_to_json_bytes() (*in module wacryptolib.utilities*), 66
dump_to_json_file() (*in module wacryptolib.utilities*), 66
dump_to_json_str() (*in module wacryptolib.utilities*), 66

E

encrypt_bytestring() (*in module wacryptolib.cipher*), 59
encrypt_file() (*wacryptolib.cryptainer.CryptainerStorage method*), 47
encrypt_payload_and_stream_cryptainer_to_filesystem() (*in module wacryptolib.cryptainer*), 44
encrypt_payload_into_cryptainer() (*in module wacryptolib.cryptainer*), 43
EncryptionError, 68
enqueue_file_for_encryption() (*wacryptolib.cryptainer.CryptainerStorage method*), 47
ExistenceError, 67
export_foreign_keystore_to_keystore_tree() (*wacryptolib.keystore.FilesystemKeystorePool method*), 57
export_to_keystore_tree() (*wacryptolib.keystore.FilesystemKeystore method*), 55
extract_metadata_from_cryptainer() (*in module wacryptolib.cryptainer*), 44

F

fetch_public_key() (*wacryptolib.trustee.TrusteeApi method*), 48
FilesystemKeystore (*class in wacryptolib.keystore*), 55
FilesystemKeystorePool (*class in wacryptolib.keystore*), 57
finalize_tarfile() (*wacryptolib.sensor.TarfileRecordAggregator method*), 64
flightbox command line option
    --cryptainer-storage, 29
    --gateway-url, 29
    --keystore-pool, 29
    --verbosity, 29
    -c, 29
    -g, 29
    -k, 29
    -v, 29
flightbox-authenticator-create command line option
    --keypair-count, 30
    --owner, 30
    --passphrase-hint, 30
    AUTHENTICATOR_DIR, 30
flightbox-authenticator-delete command line option
    AUTHENTICATOR_DIR, 30
flightbox-authenticator-validate command line option
    AUTHENTICATOR_DIR, 30
flightbox-authenticator-view command line option
    --format, 31
    -f, 31
    AUTHENTICATOR_DIR, 31
flightbox-cryptainer-decrypt command line option

--output-file, 31
-o, 31
CRYPTAINER_NAME, 32
flightbox-cryptainer-delete command line option
CRYPTAINER_NAME, 32
flightbox-cryptainer-list command line option
--format, 32
-f, 32
flightbox-cryptainer-purge command line option
--max-age, 32
--max-count, 32
--max-quota, 32
flightbox-cryptainer-summarize command line option
CRYPTAINER_NAME, 33
flightbox-cryptainer-validate command line option
CRYPTAINER_NAME, 33
flightbox-cryptoconf-generate-simple command line option
--keychain-uid, 33
flightbox-cryptoconf-generate-simple-add-key-cipher-layer command line option
--asym-cipher-algo, 34
--keychain-uid, 34
--keystore-uid, 34
--sym-cipher-algo, 34
--trustee-type, 34
flightbox-cryptoconf-generate-simple-add-key-shard command line option
--asym-cipher-algo, 34
--keychain-uid, 35
--keystore-uid, 34
--sym-cipher-algo, 35
--trustee-type, 34
flightbox-cryptoconf-generate-simple-add-key-shared-secret command line option
--threshold, 35
flightbox-cryptoconf-generate-simple-add-payload-cipher-layer command line option
--sym-cipher-algo, 35
flightbox-cryptoconf-summarize command line option
CRYPTOCONF_FILE, 36
flightbox-cryptoconf-validate command line option
CRYPTOCONF_FILE, 36
flightbox-encrypt command line option
--bundle, 36
--cryptoconf, 36
--output-basename, 36
-c, 36
-o, 36
INPUT_FILE, 36
flightbox-foreign-keystore-delete command line option
KEYSTORE_UID, 37
flightbox-foreign-keystore-import command line option
--from-gateway, 37
--from-path, 37
--from-usb, 37
--include-private-keys, 37
flightbox-foreign-keystore-list command line option
--format, 37
-f, 37
flightbox-foreign-keystore-view command line option
--format, 38
-f, 38
KEYSTORE_UID, 38
flush_dataset() (*wacryptolib.sensor.JsonDataAggregator method*), 64
FunctionalError, 67

G

gather_exception_subclasses() (*in module wacryptolib.error_handling*), 69
gather_exception_subclasses() (*wacryptolib.error_handling.StatusSlugMapper static method*), 68
gather_trustee_dependencies() (*in module wacryptolib.cryptainer*), 47
generate_keypair() (*in module wacryptolib.keygen*), 50
generate_symkey() (*in module wacryptolib.keygen*), 51
generate_uuid0() (*in module wacryptolib.utilities*), 66
get_all_foreign_keystore_metadata() (*wacryptolib.keystore.FilesystemKeystorePool method*), 57
get_closest_exception_class_for_status_slugs() (*in module wacryptolib.error_handling*), 69
get_closest_exception_class_for_status_slugs() (*wacryptolib.error_handling.StatusSlugMapper method*), 68
get_cryptainer_size_on_filesystem() (*in module wacryptolib.cryptainer*), 45
get_cryptoconf_summary() (*in module wacryptolib.cryptainer*), 44
get_foreign_keystore() (*wacryptolib.keystore.FilesystemKeystorePool method*), 58

get_foreign_keystore_metadata() (*wacryptolib.keystore.FilesystemKeystorePool method*), 58
get_free_keypairs_count() (*wacryptolib.keystore.FilesystemKeystore method*), 55
get_free_keypairs_count() (*wacryptolib.keystore.InMemoryKeystore method*), 53
get_keystore_metadata() (*wacryptolib.keystore.FilesystemKeystore method*), 55
get_local_keyfactory() (*wacryptolib.keystore.FilesystemKeystorePool method*), 58
get_message_signature() (*wacryptolib.trustee.TrusteeApi method*), 48
get_private_key() (*wacryptolib.keystore.FilesystemKeystore method*), 55
get_private_key() (*wacryptolib.keystore.InMemoryKeystore method*), 53
get_public_key() (*wacryptolib.keystore.FilesystemKeystore method*), 56
get_public_key() (*wacryptolib.keystore.InMemoryKeystore method*), 53
get_trustee_proxy() (*in module wacryptolib.cryptainer*), 47

H

hash_message() (*in module wacryptolib.utilities*), 66

I

import_foreign_keystore_from_keystore_tree() (*wacryptolib.keystore.FilesystemKeystorePool method*), 58
import_from_keystore_tree() (*wacryptolib.keystore.FilesystemKeystore method*), 56
initialize_authenticator() (*in module wacryptolib.authenticator*), 49
InMemoryKeystore (*class in wacryptolib.keystore*), 52
InMemoryKeystorePool (*class in wacryptolib.keystore*), 57
INPUT_FILE
    flightbox-encrypt command line option, 36
is_authenticator_initialized() (*in module wacryptolib.authenticator*), 49

J

join() (*wacryptolib.sensor.SensorManager method*), 64
JsonDataAggregator (*class in wacryptolib.sensor*), 64
JsonRpcProxy (*class in wacryptolib.jsonrpc_client*), 65

K

KeyAlreadyExists, 67
KeyDoesNotExist, 67
KeyLoadingError, 68
KEYSTORE_UID
    flightbox-foreign-keystore-delete command line option, 37
    flightbox-foreign-keystore-view command line option, 38
KeystoreAlreadyExists, 67
KeystoreDoesNotExist, 67
KeystoreMetadataDoesNotExist, 67

L

list_available_authdevices() (*in module wacryptolib.authdevice*), 50
list_cryptainer_names() (*wacryptolib.cryptainer.ReadonlyCryptainerStorage method*), 46
list_cryptainer_properties() (*wacryptolib.cryptainer.ReadonlyCryptainerStorage method*), 46
list_foreign_keystore_uids() (*wacryptolib.keystore.FilesystemKeystorePool method*), 58
list_keypair_identifiers() (*wacryptolib.keystore.FilesystemKeystore method*), 56
list_keypair_identifiers() (*wacryptolib.keystore.InMemoryKeystore method*), 54
load_asymmetric_key_from_pem_bytestring() (*in module wacryptolib.keygen*), 51
load_cryptainer_from_filesystem() (*in module wacryptolib.cryptainer*), 45
load_cryptainer_from_storage() (*wacryptolib.cryptainer.ReadonlyCryptainerStorage method*), 46
load_from_json_bytes() (*in module wacryptolib.utilities*), 66
load_from_json_file() (*in module wacryptolib.utilities*), 66
load_from_json_str() (*in module wacryptolib.utilities*), 66
load_keystore_metadata() (*in module wacryptolib.keystore*), 52

M

module
    wacryptolib.exceptions, 67

O

OperationNotSupported, 68

P

PayloadEncryptionPipeline (*class in wacryptolib.cipher*), 59
PeriodicTaskHandler() (*in module wacryptolib.utilities*), 65
PeriodicValuePoller (*class in wacryptolib.sensor*), 64

R

read_tarfile_from_bytestring() (*wacryptolib.sensor.TarfileRecordAggregator static method*), 64
ReadonlyCryptainerStorage (*class in wacryptolib.cryptainer*), 45
ReadonlyTrusteeApi (*class in wacryptolib.trustee*), 49
recombine_chunks() (*in module wacryptolib.utilities*), 67
recombine_secret_from_shards() (*in module wacryptolib.shared_secret*), 63
request_decryption_authorization() (*wacryptolib.trustee.TrusteeApi method*), 48
request_decryption_authorizations() (*in module wacryptolib.cryptainer*), 48

S

SchemaValidationError, 68
SensorManager (*class in wacryptolib.sensor*), 64
set_keypair() (*wacryptolib.keystore.FilesystemKeystore method*), 56
set_keypair() (*wacryptolib.keystore.InMemoryKeystore method*), 54
set_private_key() (*wacryptolib.keystore.FilesystemKeystore method*), 57
set_private_key() (*wacryptolib.keystore.InMemoryKeystore method*), 54
set_public_key() (*wacryptolib.keystore.FilesystemKeystore method*), 57
set_public_key() (*wacryptolib.keystore.InMemoryKeystore method*), 54
sign_message() (*in module wacryptolib.signature*), 58
SignatureCreationError, 68
SignatureVerificationError, 68
slugify_exception_class() (*in module wacryptolib.error_handling*), 69
slugify_exception_class() (*wacryptolib.error_handling.StatusSlugMapper method*), 69
split_as_chunks() (*in module wacryptolib.utilities*), 66
split_secret_into_shards() (*in module wacryptolib.shared_secret*), 63
start() (*wacryptolib.sensor.SensorManager method*), 64
status_slugs_response_error_handler() (*in module wacryptolib.jsonrpc_client*), 65
StatusSlugMapper (*class in wacryptolib.error_handling*), 68
stop() (*wacryptolib.sensor.SensorManager method*), 64
STREAMABLE_CIPHER_ALGOS (*in module wacryptolib.cipher*), 59
SUPPORTED_ASYMMETRIC_KEY_ALGOS (*in module wacryptolib.keygen*), 50
SUPPORTED_CIPHER_ALGOS (*in module wacryptolib.cipher*), 59
SUPPORTED_HASH_ALGOS (*in module wacryptolib.utilities*), 66
SUPPORTED_SIGNATURE_ALGOS (*in module wacryptolib.signature*), 58
SUPPORTED_SYMMETRIC_KEY_ALGOS (*in module wacryptolib.keygen*), 51

T

TarfileRecordAggregator (*class in wacryptolib.sensor*), 63
TaskRunnerStateMachineBase() (*in module wacryptolib.utilities*), 65
TrusteeApi (*class in wacryptolib.trustee*), 48

V

ValidationError, 68
verify_message_signature() (*in module wacryptolib.signature*), 58

W

wacryptolib.exceptions
module, 67
wait_for_idle_state() (*wacryptolib.cryptainer.CryptainerStorage method*), 47