PyKeePass

Oct 08, 2020

Contents

CHAPTER 1

pykeepass

class pykeepass.pykeepass.**PyKeePass**(*filename*, *password=None*, *keyfile=None*, *transformed_key=None*)

Open a KeePass database

Args:

filename (`str`, optional): path to database or stream object. If None, the path given when the database was opened is used.

password (`str`, optional): database password. If None, database is assumed to have no password

keyfile (`str`, optional): path to keyfile. If None, database is assumed to have no keyfile

transformed_key (`bytes`, optional): precomputed transformed key.

Raises:

CredentialsError: raised when password/keyfile or transformed key are wrong

HeaderChecksumError: raised when checksum in database header is is wrong. e.g. database tampering or file corruption

PayloadChecksumError: raised when payload blocks checksum is wrong, e.g. corruption during database saving

Todo:

- raise, no filename provided, database not open

`dump_xml`(*filename*)

Dump the contents of the database to file as XML

Args: filename (str): path to output file

`encryption_algorithm`

str: encryption algorithm used by database during decryption. Can be one of 'aes256', 'chacha20', or 'twofish'.

`entries`

`list` of `Entry`: list of all Entry objects in database, excluding history

groups
`list` of `Group`: list of all Group objects in database

kdf_algorithm
str: key derivation algorithm used by database during decryption. Can be one of 'aeskdf', 'argon2', or 'aeskdf'

read(*filename=None*, *password=None*, *keyfile=None*, *transformed_key=None*)
See class docstring.

Todo:

- raise, no filename provided, database not open

root_group
Group: root Group of database

save(*filename=None*, *transformed_key=None*)
Save current database object to disk.

Args:

filename (`str`, optional): path to database or stream object. If None, the path given when the database was opened is used.

transformed_key (`bytes`, optional): precomputed transformed key.

transformed_key
bytes: transformed key used in database decryption. May be cached and passed to *open* for faster database opening

tree
lxml.etree._ElementTree: database XML payload

version
tuple: Length 2 tuple of ints containing major and minor versions. Generally (3, 1) or (4, 0).

xml()
Get XML part of database as string

Returns: str: XML payload section of database.

CHAPTER 2

group

class pykeepass.group.**Group**(*name=None*, *element=None*, *icon=None*, *notes=None*, *kp=None*, *expires=None*, *expiry_time=None*)

CHAPTER 3

entry

class pykeepass.entry.**Entry**(*title=None*, *username=None*, *password=None*, *url=None*, *notes=None*, *tags=None*, *expires=False*, *expiry_time=None*, *icon=None*, *autotype_sequence=None*, *autotype_enabled=True*, *element=None*, *kp=None*)

ref(*attribute*)
Create reference to an attribute of this element.

save_history()
Save the entry in its history

touch(*modify=False*)
Update last access time of an entry

CHAPTER 4

attachment

CHAPTER 5

icons

CHAPTER 6

exceptions

exception pykeepass.exceptions.**BinaryError**

exception pykeepass.exceptions.**CredentialsError**

exception pykeepass.exceptions.**HeaderChecksumError**

exception pykeepass.exceptions.**PayloadChecksumError**

CHAPTER 7

baseelement

class pykeepass.baseelement.**BaseElement**(*element=None*, *kp=None*, *icon=None*, *expires=False*, *expiry_time=None*)

Entry and Group inherit from this class

uuid

Returns uuid of this element as a uuid.UUID object

CHAPTER 8

kdbx_parsing.common

class pykeepass.kdbx_parsing.common.**AES256Payload**(*subcon*)

class pykeepass.kdbx_parsing.common.**ARCFourVariantStream**(*protected_stream_key*, *subcon*)

class pykeepass.kdbx_parsing.common.**ChaCha20Payload**(*subcon*)

class pykeepass.kdbx_parsing.common.**ChaCha20Stream**(*protected_stream_key*, *subcon*)

class pykeepass.kdbx_parsing.common.**Concatenated**(*subcon*)
Data Blocks <—> Bytes

exception pykeepass.kdbx_parsing.common.**CredentialsError**

class pykeepass.kdbx_parsing.common.**Decompressed**(*subcon*)
Compressed Bytes <—> Decompressed Bytes

class pykeepass.kdbx_parsing.common.**DecryptedPayload**(*subcon*)
Encrypted Bytes <—> Decrypted Bytes

class pykeepass.kdbx_parsing.common.**DynamicDict**(*key*, *subcon*, *lump=[]*)
ListContainer <—> Container Convenience mapping so we dont have to iterate ListContainer to find the right item

FIXME: lump kwarg was added to get around the fact that InnerHeader is not truly a dict. We lump all 'binary' InnerHeaderItems into a single list

exception pykeepass.kdbx_parsing.common.**HeaderChecksumError**

exception pykeepass.kdbx_parsing.common.**PayloadChecksumError**

class pykeepass.kdbx_parsing.common.**Salsa20Stream**(*protected_stream_key*, *subcon*)

class pykeepass.kdbx_parsing.common.**TwoFishPayload**(*subcon*)

pykeepass.kdbx_parsing.common.**Unprotect**(*protected_stream_id*, *protected_stream_key*, *subcon*)
Select stream cipher based on protected_stream_id

class pykeepass.kdbx_parsing.common.**UnprotectedStream**(*protected_stream_key*, *subcon*)
lxml etree <—> unprotected lxml etree Iterate etree for Protected elements and decrypt using cipher provided by get_cipher

class pykeepass.kdbx_parsing.common.**XML**(*subcon*)
Bytes <—> lxml etree

pykeepass.kdbx_parsing.common.**aes_kdf**(*key*, *rounds*, *key_composite*)
Set up a context for AES128-ECB encryption to find transformed_key

pykeepass.kdbx_parsing.common.**compute_key_composite**(*password=None*, *keyfile=None*)
Compute composite key. Used in header verification and payload decryption.

pykeepass.kdbx_parsing.common.**compute_master**(*context*)
Computes master key from transformed key and master seed. Used in payload decryption.

CHAPTER 9

kdbx_parsing.kdbx

CHAPTER 10

kdbx_parsing.kdbx3

pykeepass.kdbx_parsing.kdbx3.**compute_transformed**(*context*)
Compute transformed key for opening database

CHAPTER 11

kdbx_parsing.kdbx4

pykeepass.kdbx_parsing.kdbx4.**compute_header_hmac_hash**(*context*)
Compute HMAC-SHA256 hash of header. Used to prevent header tampering.

pykeepass.kdbx_parsing.kdbx4.**compute_payload_block_hash**(*this*)
Compute hash of each payload block. Used to prevent payload corruption and tampering.

pykeepass.kdbx_parsing.kdbx4.**compute_transformed**(*context*)
Compute transformed key for opening database

CHAPTER 12

kdbx_parsing.pytwofish

CHAPTER 13

kdbx_parsing.twofish

pykeepass.kdbx_parsing.twofish.**Twofish**
alias of pykeepass.kdbx_parsing.twofish.python_Twofish

This library allows you to write entries to a KeePass database.

Come chat at #pykeepass on Freenode or #pykeepass:matrix.org on Matrix.

CHAPTER 14

Example

```
from pykeepass import PyKeePass

# load database
>>> kp = PyKeePass('db.kdbx', password='somePassw0rd')

# find any group by its name
>>> group = kp.find_groups(name='social', first=True)

# get the entries in a group
>>> group.entries
[Entry: "social/facebook (myusername)", Entry: "social/twitter (myusername)"]

# find any entry by its title
>>> entry = kp.find_entries(title='facebook', first=True)

# retrieve the associated password
>>> entry.password
's3cure_p455w0rd'

# update an entry
>>> entry.notes = 'primary facebook account'

# create a new group
>>> group = kp.add_group(kp.root_group, 'email')

# create a new entry
>>> kp.add_entry(group, 'gmail', 'myusername', 'myPassw0rdXX')
Entry: "email/gmail (myusername)"

# save database
>>> kp.save()
```

CHAPTER 15

Finding Entries

find_entries (title=None, username=None, password=None, url=None, notes=None, path=None, uuid=None, tags=None, string=None, group=None, recursive=True, regex=False, flags=None, history=False, first=False)

Returns entries which match all provided parameters, where `title`, `username`, `password`, `url`, `notes`, `path`, and `autotype_sequence` are strings, `string` is a dict, `autotype_enabled` is a boolean, `uuid` is a `uuid.UUID` and `tags` is a list of strings. This function has optional `regex` boolean and `flags` string arguments, which means to interpret search strings as XSLT style regular expressions with flags.

The `path` string is a full path to an entry (ex. `'foobar_group/foobar_entry'`). This implies `first=True`. All other arguments are ignored when this is given. This is useful for handling user input.

The `string` dict allows for searching custom string fields. ex. `{'custom_field1': 'custom value', 'custom_field2': 'custom value'}`

The `group` argument determines what `Group` to search under, and the `recursive` boolean controls whether to search recursively.

The `history` (default `False`) boolean controls whether history entries should be included in the search results.

The `first` (default `False`) boolean controls whether to return the first matched item, or a list of matched items.

- if `first=False`, the function returns a list of `Entry` s or `[]` if there are no matches
- if `first=True`, the function returns the first `Entry` match, or `None` if there are no matches

entries

a flattened list of all entries in the database

```
>>> kp.entries
[Entry: "foo_entry (myusername)", Entry: "foobar_entry (myusername)", Entry: "social/
↪gmail (myusername)", Entry: "social/facebook (myusername)"]

>>> kp.find_entries(title='gmail', first=True)
Entry: "social/gmail (myusername)"

>>> kp.find_entries(title='foo.*', regex=True)
```

(continues on next page)

(continued from previous page)

```
[Entry: "foo_entry (myusername)", Entry: "foobar_entry (myusername)"]

>>> entry = kp.find_entries(title='foo.*', url='.*facebook.*', regex=True, first=True)
>>> entry.url
'facebook.com'
>>> entry.title
'foo_entry'

>>> group = kp.find_group(name='social', first=True)
>>> kp.find_entries(title='facebook', group=group, recursive=False, first=True)
Entry: "social/facebook (myusername)"
```

CHAPTER 16

Finding Groups

find_groups (name=None, path=None, uuid=None, notes=None, group=None, recursive=True, regex=False, flags=None, first=False)

where `name`, `path`, and `notes` are strings, `uuid` is a `uuid.UUID`. This function has optional `regex` boolean and `flags` string arguments, which means to interpret search strings as XSLT style regular expressions with flags.

The `path` string is a full path to a group (ex. `'foobar_group/sub_group'`). This implies `first=True`. All other arguments are ignored when this is given. This is useful for handling user input.

The `group` argument determines what `Group` to search under, and the `recursive` boolean controls whether to search recursively.

The `first` (default `False`) boolean controls whether to return the first matched item, or a list of matched items.

- if `first=False`, the function returns a list of `Group` s or `[]` if there are no matches
- if `first=True`, the function returns the first `Group` match, or `None` if there are no matches

root_group

the `Root` group to the database

groups

a flattened list of all groups in the database

```
>>> kp.groups
[Group: "foo", Group "foobar", Group: "social", Group: "social/foo_subgroup"]

>>> kp.find_groups(name='foo', first=True)
Group: "foo"

>>> kp.find_groups(name='foo.*', regex=True)
[Group: "foo", Group "foobar"]

>>> kp.find_groups(path='social/', regex=True)
[Group: "social", Group: "social/foo_subgroup"]
```

(continues on next page)

(continued from previous page)

```
>>> kp.find_groups(name='social', first=True).subgroups
[Group: "social/foo_subgroup"]

>>> kp.root_group
Group: "/"
```

CHAPTER 17

Adding Entries

add_entry (destination_group, title, username, password, url=None, notes=None, tags=None, expiry_time=None, icon=None, force_creation=False)

delete_entry (entry)

move_entry (entry, destination_group)

where `destination_group` is a `Group` instance. `entry` is an `Entry` instance. `title`, `username`, `password`, `url`, `notes`, `tags`, `icon` are strings. `expiry_time` is a `datetime` instance.

If `expiry_time` is a naive datetime object (i.e. `expiry_time.tzinfo` is not set), the timezone is retrieved from `dateutil.tz.gettz()`.

```
# add a new entry to the Root group
>>> kp.add_entry(kp.root_group, 'testing', 'foo_user', 'passw0rd')
Entry: "testing (foo_user)"

# add a new entry to the social group
>>> group = find_groups(name='social', first=True)
>>> entry = kp.add_entry(group, 'testing', 'foo_user', 'passw0rd')
Entry: "testing (foo_user)"

# save the database
>>> kp.save()

# delete an entry
>>> kp.delete_entry(entry)

# move an entry
>>> kp.move_entry(entry, kp.root_group)

# save the database
>>> kp.save()
```

CHAPTER 18

Adding Groups

add_group (destination_group, group_name, icon=None, notes=None)

delete_group (group)

move_group (group, destination_group)

`destination_group` and `group` are instances of `Group`. `group_name` is a string

```
# add a new group to the Root group
>>> group = kp.add_group(kp.root_group, 'social')

# add a new group to the social group
>>> group2 = kp.add_group(group, 'gmail')
Group: "social/gmail"

# save the database
>>> kp.save()

# delete a group
>>> kp.delete_group(group)

# move a group
>>> kp.move_group(group2, kp.root_group)

# save the database
>>> kp.save()
```

CHAPTER 19

Attachments

In this section, *binary* refers to the bytes of the attached data (stored at the root level of the database), while *attachment* is a reference to a binary (stored in an entry). A binary can have none, one or many attachments.

add_binary (data, compressed=True, protected=True)

where `data` is bytes. Adds a blob of data to the database. The attachment reference must still be added to an entry (see below). `compressed` only applies to KDBX3 and `protected` only applies to KDBX4. Returns id of attachment.

delete_binary (id)

where `id` is an int. Removes binary data from the database and deletes any attachments that reference it. Since attachments reference binaries by their positional index, attachments that reference binaries with id > `id` will automatically be decremented.

find_attachments (id=None, filename=None, element=None, recursive=True, regex=False, flags=None, history=False, first=False)

where `id` is an int, `filename` is a string, and element is an `Entry` or `Group` to search under.

- if `first=False`, the function returns a list of `Attachment` s or `[]` if there are no matches
- if `first=True`, the function returns the first `Attachment` match, or `None` if there are no matches

binaries

list of bytestrings containing binary data. List index corresponds to attachment id.

attachments

list containing all `Attachment` s in the database.

Entry.add_attachment (id, filename)

where `id` is an int and `filename` is a string. Creates a reference using the given filename to a database binary. The existence of a binary with the given id is not checked. Returns `Attachment`.

Entry.delete_attachment (attachment)

where `attachment` is an `Attachment`. Deletes a reference to a database binary.

Entry.attachments

list of `Attachment` s for this Entry.

Attachment.id

id of data that this attachment points to

Attachment.filename

string representing this attachment

Attachment.data

the data that this attachment points to. Raises `BinaryError` if data does not exist.

Attachment.entry

the entry that this attachment is attached to

```
>>> e = kp.add_entry(kp.root_group, title='foo', username='', password='')

# add attachment data to the db
>>> binary_id = kp.add_binary(b'Hello world')

>>> kp.binaries
[b'Hello world']

# add attachment reference to entry
>>> a = e.add_attachment(binary_id, 'hello.txt')
>>> a
Attachment: 'hello.txt' -> 0

# access attachments
>>> a
Attachment: 'hello.txt' -> 0
>>> a.id
0
>>> a.filename
'hello.txt'
>>> a.data
b'Hello world'
>>> e.attachments
[Attachment: 'hello.txt' -> 0]

# list all attachments in the database
>>> kp.attachments
[Attachment: 'hello.txt' -> 0]

# search attachments
>>> kp.find_attachments(filename='hello.txt')
[Attachment: 'hello.txt' -> 0]

# delete attachment reference
>>> e.delete_attachment(a)

# or, delete both attachment reference and binary
>>> kp.delete_binary(binary_id)
```

CHAPTER 20

Miscellaneous

read (filename=None, password=None, keyfile=None, transformed_key=None)

where `filename`, `password`, and `keyfile` are strings. `filename` is the path to the database, `password` is the master password string, and `keyfile` is the path to the database keyfile. At least one of `password` and `keyfile` is required. Alternatively, the derived key can be supplied directly through `transformed_key`.

Can raise `CredentialsError`, `HeaderChecksumError`, or `PayloadChecksumError`.

save (filename=None)

where `filename` is the path of the file to save to. If `filename` is not given, the path given in `read` will be used.

password

string containing database password. Can also be set. Use `None` for no password.

keyfile

string containing path to the database keyfile. Can also be set. Use `None` for no keyfile.

version

tuple containing database version. e.g. `(3, 1)` is a KDBX version 3.1 database.

encryption_algorithm

string containing algorithm used to encrypt database. Possible values are `aes256`, `chacha20`, and `twofish`.

create_database (filename, password=None, keyfile=None, transformed_key=None)

create a new database at `filename` with supplied credentials. Returns `PyKeePass` object

CHAPTER 21

Tests

To run them issue `python -m unittest discover` in the repository.

CHAPTER 22

Indices and tables

- genindex
- modindex
- search

Python Module Index

p

Index

R

S

T

U

V

X