PyJKS Documentation

Release 20.0.0

Kurt Rose and contributors

May 23, 2020

Contents

PyJKS is *the* pure-Python library for Java KeyStore (JKS) parsing, decryption, and manipulation. PyJKS supports vanilla JKS, JCEKS, BKS, and UBER (BouncyCastle) keystore formats.

In the past, Python projects relied on external tools (*keytool*), intermediate formats (*PKCS12* and *PEM*), and the JVM to work with encrypted material locked within JKS files. Now, PyJKS changes that.

CHAPTER 1

Examples

See the *Examples* page for usage examples of PyJKS.

CHAPTER 2

Installation

You can install `pyjks` with `pip`:

```
$ pip install pyjks
```

If you receive an error like:

```
error: Microsoft Visual C++ 14.0 is required. Get it with "Microsoft Visual C++ Build␣
↪Tools": https://visualstudio.microsoft.com/downloads/
```

on Windows you will need to download the Visual C++ build tools by visiting https://visualstudio.microsoft.com/thank-you-downloading-visual-studio/?sku=BuildTools&rel=16

Save the file, then run it. Choose "Workloads" tab, then select the "C++ build tools". Under the "Optional" installed items, be certain to select all of `MSVC vxxx - VS 2019 C++ build tools`, `Windows 10 SDK` (latest version), and `C++/CLI support for build tools`. Reboot, then run the `pip` command again.

CHAPTER 3

Contents:

3.1 Examples

Building an OpenSSL context using a JKS through PyJKS:

```python
import jks
import OpenSSL

ASN1 = OpenSSL.crypto.FILETYPE_ASN1

def jksfile2context(jks_file, passphrase, key_alias, key_password=None):

    keystore = jks.KeyStore.load(jks_file, passphrase)
    pk_entry = keystore.private_keys[key_alias]

    # if the key could not be decrypted using the store password,
    # decrypt with a custom password now
    if not pk_entry.is_decrypted():
        pk_entry.decrypt(key_password)

    pkey = OpenSSL.crypto.load_privatekey(ASN1, pk_entry.pkey)
    public_cert = OpenSSL.crypto.load_certificate(ASN1, pk_entry.cert_chain[0][1])
    trusted_certs = [OpenSSL.crypto.load_certificate(ASN1, cert.cert)
                     for alias, cert in keystore.certs]

    ctx = OpenSSL.SSL.Context(OpenSSL.SSL.TLSv1_METHOD)
    ctx.use_privatekey(pkey)
    ctx.use_certificate(public_cert)
    ctx.check_privatekey() # want to know ASAP if there is a problem
    cert_store = ctx.get_cert_store()
    for cert in trusted_certs:
        cert_store.add_cert(cert)

    return ctx
```

Reading a JKS or JCEKS keystore and dumping out its contents in the PEM format:

```
from __future__ import print_function
import sys, base64, textwrap
import jks

def print_pem(der_bytes, type):
    print("-----BEGIN %s-----" % type)
    print("\r\n".join(textwrap.wrap(base64.b64encode(der_bytes).decode('ascii'), 64)))
    print("-----END %s-----" % type)

ks = jks.KeyStore.load("keystore.jks", "XXXXXXXX")
# if any of the keys in the store use a password that is not the same as the store␣
↪password:
# ks.entries["key1"].decrypt("key_password")

for alias, pk in ks.private_keys.items():
    print("Private key: %s" % pk.alias)
    if pk.algorithm_oid == jks.util.RSA_ENCRYPTION_OID:
        print_pem(pk.pkey, "RSA PRIVATE KEY")
    else:
        print_pem(pk.pkey_pkcs8, "PRIVATE KEY")

    for c in pk.cert_chain:
        print_pem(c[1], "CERTIFICATE")
    print()

for alias, c in ks.certs.items():
    print("Certificate: %s" % c.alias)
    print_pem(c.cert, "CERTIFICATE")
    print()

for alias, sk in ks.secret_keys.items():
    print("Secret key: %s" % sk.alias)
    print("  Algorithm: %s" % sk.algorithm)
    print("  Key size: %d bits" % sk.key_size)
    print("  Key: %s" % "".join("{:02x}".format(b) for b in bytearray(sk.key)))
    print()
```

Generating a basic self signed certificate with OpenSSL and saving it in a jks keystore:

```
import OpenSSL
import jks

# generate key
key = OpenSSL.crypto.PKey()
key.generate_key(OpenSSL.crypto.TYPE_RSA, 2048)

# generate a self signed certificate
cert = OpenSSL.crypto.X509()
cert.get_subject().CN = 'my.server.example.com'
cert.set_serial_number(473289472)
cert.gmtime_adj_notBefore(0)
cert.gmtime_adj_notAfter(365*24*60*60)
cert.set_issuer(cert.get_subject())
cert.set_pubkey(key)
cert.sign(key, 'sha256')
```

(continues on next page)

(continued from previous page)

```
# dumping the key and cert to ASN1
dumped_cert = OpenSSL.crypto.dump_certificate(OpenSSL.crypto.FILETYPE_ASN1, cert)
dumped_key = OpenSSL.crypto.dump_privatekey(OpenSSL.crypto.FILETYPE_ASN1, key)

# creating a private key entry
pke = jks.PrivateKeyEntry.new('self signed cert', [dumped_cert], dumped_key, 'rsa_raw
↪')

# if we want the private key entry to have a unique password, we can encrypt it␣
↪beforehand
# if it is not ecrypted when saved, it will be encrypted with the same password as␣
↪the keystore
#pke.encrypt('my_private_key_password')

# creating a jks keystore with the private key, and saving it
keystore = jks.KeyStore.new('jks', [pke])
keystore.save('./my_keystore.jks', 'my_password')
```

3.2 Concepts

3.2.1 Store and entry passwords

Java keystores usually involve two kinds of passwords:

- Passwords to protect individual key entries
- A password to protect the integrity of the keystore as a whole

These passwords serve different purposes: the individual key passwords serve as secret material for encrypting the entries with a PBE algorithm (Password-Based Encryption). The store password is typically used to detect tampering of the store by using it as part of the input to a cryptographic hash calculation or as a key for a MAC.

In the general case, each entry in the store can have a different password associated with it, with an additional final password being used for the keystore integrity check. To reduce the amount of passwords that needs to be kept track of though, it is common for a single password to be used for both the store integrity as well as all individual key entries.

To support the common case where key entries are protected using the store password, the `load` and `loads` class functions exposed by the different supported store types in pyjks contain a `try_decrypt_keys` keyword argument.

If set to `True`, the function will automatically try to decrypt each key entry it encounters using the store password. Any entry that fails to decrypt with the store password must therefore have been stored using a different password, and is left alone for the user to manually call `decrypt()` on afterwards.

3.2.2 Store types

JKS:

- Key protection algorithm: proprietary JavaSoft algorithm (1.3.6.1.4.1.42.2.17.1.1)
- Store signature algorithm: SHA-1 hash

JCEKS:

- Key protection algorithm: proprietary PBE_WITH_MD5_AND_DES3_CBC (1.3.6.1.4.1.42.2.19.1)
- Store signature algorithm: SHA-1 hash

BKS:

- Key protection algorithm: PBEWithSHAAnd3KeyTripleDESCBC (1.2.840.113549.1.12.1.3)
- Store signature algorithm: HMAC-SHA1

UBER:

- Key protection algorithm: PBEWithSHAAnd3KeyTripleDESCBC (1.2.840.113549.1.12.1.3)
- Store signature algorithm: SHA-1 hash
- Store encryption algorithm: PBEWithSHAAndTwofishCBC (unknown OID, proprietary?)

3.3 JKS and JCEKS keystores

3.3.1 Background

The JKS keystore format is the format that originally shipped with Java. It is implemented by the traditional "Sun" cryptography provider.

JCEKS is an improved keystore format introduced with the Java Cryptography Extension (JCE). It is implemented by the SunJCE cryptography provider.

JCEKS keystores improve upon JKS keystores in 2 ways:

- A stronger key protection algorithm is used
- They allow for arbitrary (symmetric) secret keys to be stored (e.g. AES, DES, etc.)

3.3.2 Store types

class `jks.jks.`**`KeyStore`**(*store_type*, *entries*)

Bases: `jks.util.AbstractKeystore`

Represents a loaded JKS or JCEKS keystore.

`entries`

A dictionary of all entries in the keystore, mapped by alias.

`store_type`

A string indicating the type of keystore that was loaded. Can be one of `jks`, `jceks`.

`classmethod load`(*filename*, *store_password*, *try_decrypt_keys=True*)

Convenience wrapper function; reads the contents of the given file and passes it through to *`loads()`*. See *`loads()`*.

`classmethod loads`(*data*, *store_password*, *try_decrypt_keys=True*)

Loads the given keystore file using the supplied password for verifying its integrity, and returns a *`KeyStore`* instance.

Note that entries in the store that represent some form of cryptographic key material are stored in encrypted form, and therefore require decryption before becoming accessible.

Upon original creation of a key entry in a Java keystore, users are presented with the choice to either use the same password as the store password, or use a custom one. The most common choice is to use the store password for the individual key entries as well.

For ease of use in this typical scenario, this function will attempt to decrypt each key entry it encounters with the store password:

- If the key can be successfully decrypted with the store password, the entry is returned in its decrypted form, and its attributes are immediately accessible.
- If the key cannot be decrypted with the store password, the entry is returned in its encrypted form, and requires a manual follow-up decrypt(key_password) call from the user before its individual attributes become accessible.

Setting `try_decrypt_keys` to `False` disables this automatic decryption attempt, and returns all key entries in encrypted form.

You can query whether a returned entry object has already been decrypted by calling the `is_decrypted()` method on it. Attempting to access attributes of an entry that has not yet been decrypted will result in a *`NotYetDecryptedException`*.

Parameters

- **`data`** (*`bytes`*) – Byte string representation of the keystore to be loaded.
- **`password`** (*`str`*) – Keystore password string
- **`try_decrypt_keys`** (*`bool`*) – Whether to automatically try to decrypt any encountered key entries using the same password as the keystore password.

Returns

A loaded *`KeyStore`* instance, if the keystore could be successfully parsed and the supplied store password is correct.

If the `try_decrypt_keys` parameter was set to `True`, any keys that could be successfully decrypted using the store password have already been decrypted; otherwise, no atttempt to decrypt any key entries is made.

Raises

- ***`BadKeystoreFormatException`*** – If the keystore is malformed in some way
- ***`UnsupportedKeystoreVersionException`*** – If the keystore contains an unknown format version number
- ***`KeystoreSignatureException`*** – If the keystore signature could not be verified using the supplied store password
- ***`DuplicateAliasException`*** – If the keystore contains duplicate aliases

`classmethod new`(*store_type*, *store_entries*)

Helper function to create a new KeyStore.

Parameters

- **`store_type`** (*`string`*) – What kind of keystore the store should be. Valid options are jks or jceks.
- **`store_entries`** (*`list`*) – Existing entries that should be added to the keystore.

Returns A loaded *`KeyStore`* instance, with the specified entries.

Raises

- ***`DuplicateAliasException`*** – If some of the entries have the same alias.
- **`UnsupportedKeyStoreTypeException`** – If the keystore is of an unsupported type
- **`UnsupportedKeyStoreEntryTypeException`** – If some of the keystore entries are unsupported (in this keystore type)

save (*filename*, *store_password*)
Convenience wrapper function; calls the `saves()` and saves the content to a file.

saves (*store_password*)
Saves the keystore so that it can be read by other applications.

If any of the private keys are unencrypted, they will be encrypted with the same password as the keystore.

Parameters **store_password** (`str`) – Password for the created keystore (and for any unencrypted keys)

Returns A byte string representation of the keystore.

Raises

- `UnsupportedKeystoreTypeException` – If the keystore is of an unsupported type
- `UnsupportedKeystoreEntryTypeException` – If the keystore contains an unsupported entry type

certs
A subset of the `entries` dictionary, filtered down to only those entries of type `TrustedCertEntry`.

private_keys
A subset of the `entries` dictionary, filtered down to only those entries of type `PrivateKeyEntry`.

secret_keys
A subset of the `entries` dictionary, filtered down to only those entries of type `SecretKeyEntry`.

3.3.3 Entry types

class `jks.jks.`**TrustedCertEntry** (***kwargs*)
Bases: `jks.util.AbstractKeystoreEntry`

Represents a trusted certificate entry in a JKS or JCEKS keystore.

decrypt (*key_password*)
Does nothing for this entry type; certificates are inherently public data and are not stored in encrypted form.

encrypt (*key_password*)
Does nothing for this entry type; certificates are inherently public data and are not stored in encrypted form.

is_decrypted ()
Always returns `True` for this entry type.

classmethod new (*alias*, *cert*)
Helper function to create a new TrustedCertEntry.

Parameters

- **alias** (`str`) – The alias for the Trusted Cert Entry
- **certs** (`str`) – The certificate, as a byte string.

Returns A loaded `TrustedCertEntry` instance, ready to be placed in a keystore.

cert = None
A byte string containing the actual certificate data. In the case of X.509 certificates, this is the DER-encoded X.509 representation of the certificate.

type = None
A string indicating the type of certificate. Unless in exotic applications, this is usually `X.509`.

class jks.jks.**PrivateKeyEntry**(***kwargs*)

Bases: `jks.util.AbstractKeystoreEntry`

Represents a private key entry in a JKS or JCEKS keystore (e.g. an RSA or DSA private key).

pkey

Note: Only accessible after a call to *decrypt()*; until then, accessing this attribute will raise a *NotYetDecryptedException*. See also `try_decrypt_keys` on *jks.jks.KeyStore.loads()*.

A byte string containing the value of the `privateKey` field of the PKCS#8 `PrivateKeyInfo` representation of the private key. See RFC 5208, section 5: Private-Key Information Syntax.

pkey_pkcs8

Note: Only accessible after a call to *decrypt()*; until then, accessing this attribute will raise a *NotYetDecryptedException*. See also `try_decrypt_keys` on *jks.jks.KeyStore.loads()*.

A byte string containing the DER-encoded PKCS#8 `PrivateKeyInfo` representation of the private key. See RFC 5208, section 5: Private-Key Information Syntax.

algorithm_oid

Note: Only accessible after a call to *decrypt()*; until then, accessing this attribute will raise a *NotYetDecryptedException*. See also `try_decrypt_keys` on *jks.jks.KeyStore.loads()*.

A tuple of integers corresponding to the algorithm OID for which the private key is valid.

Common values include:

- `(1,2,840,113549,1,1,1)` (alias `rsaEncryption`)
- `(1,2,840,10040,4,1)` (alias `id-dsa`).

cert_chain = None

A list of tuples, representing the certificate chain associated with the private key. Each element of the list of a 2-tuple containing the following data:

- `[0]`: A string indicating the type of certificate. Unless in exotic applications, this is usually `X.509`.
- `[1]`: A byte string containing the actual certificate data. In the case of X.509 certificates, this is the DER-encoded X.509 representation of the certificate.

classmethod new(*alias*, *certs*, *key*, *key_format='pkcs8'*)

Helper function to create a new PrivateKeyEntry.

Parameters

- **alias** (*str*) – The alias for the Private Key Entry
- **certs** (*list*) – An list of certificates, as byte strings. The first one should be the one belonging to the private key, the others the chain (in correct order).

- **key** (*str*) – A byte string containing the private key in the format specified in the key_format parameter (default pkcs8).
- **key_format** (*str*) – The format of the provided private key. Valid options are pkcs8 or rsa_raw. Defaults to pkcs8.

Returns A loaded *PrivateKeyEntry* instance, ready to be placed in a keystore.

Raises ***UnsupportedKeyFormatException*** – If the key format is unsupported.

is_decrypted()

Returns `True` if the entry has already been decrypted, `False` otherwise.

decrypt(*key_password*)

Decrypts the entry using the given password. Has no effect if the entry has already been decrypted.

Parameters **key_password** (*str*) – The password to decrypt the entry with. If the entry was loaded from a JCEKS keystore, the password must not contain any characters outside of the ASCII character set.

Raises

- ***DecryptionFailureException*** – If the entry could not be decrypted using the given password.
- ***UnexpectedAlgorithmException*** – If the entry was encrypted with an unknown or unexpected algorithm
- **ValueError** – If the entry was loaded from a JCEKS keystore and the password contains non-ASCII characters.

encrypt(*key_password*)

Encrypts the private key, so that it can be saved to a keystore.

This will make it necessary to decrypt it again if it is going to be used later. Has no effect if the entry is already encrypted.

Parameters **key_password** (*str*) – The password to encrypt the entry with.

class jks.jks.**SecretKeyEntry**(***kwargs*)

Bases: `jks.util.AbstractKeystoreEntry`

Represents a secret (symmetric) key entry in a JCEKS keystore (e.g. an AES or DES key).

algorithm

Note: Only accessible after a call to *decrypt()*; until then, accessing this attribute will raise a *NotYetDecryptedException*. See also `try_decrypt_keys` on *jks.jks.KeyStore.loads()*.

A string containing the name of the algorithm for which the key is valid, as known to the Java cryptography provider that supplied the corresponding SecretKey object.

key

Note: Only accessible after a call to *decrypt()*; until then, accessing this attribute will raise a *NotYetDecryptedException*. See also `try_decrypt_keys` on *jks.jks.KeyStore.loads()*.

A byte string containing the raw secret key.

key_size

> **Note:** Only accessible after a call to `decrypt()`; until then, accessing this attribute will raise a `NotYetDecryptedException`. See also `try_decrypt_keys` on `jks.jks.KeyStore.loads()`.

An integer containing the size of the key, in bits. For DES and 3DES keys, the sizes 64 bits resp. 192 bits are returned.

classmethod new(*alias*, *sealed_obj*, *algorithm*, *key*, *key_size*)

Helper function to create a new SecretKeyEntry.

Returns A loaded `SecretKeyEntry` instance, ready to be placed in a keystore.

is_decrypted()

Returns `True` if the entry has already been decrypted, `False` otherwise.

decrypt(*key_password*)

Decrypts the entry using the given password. Has no effect if the entry has already been decrypted.

Parameters **key_password** (`str`) – The password to decrypt the entry with. Must not contain any characters outside of the ASCII character set.

Raises

- **`DecryptionFailureException`** – If the entry could not be decrypted using the given password.
- **`UnexpectedAlgorithmException`** – If the entry was encrypted with an unknown or unexpected algorithm
- **`ValueError`** – If the password contains non-ASCII characters.

encrypt(*key_password*)

Encrypts the Secret Key so that the keystore can be saved

3.4 BouncyCastle Keystores (BKS and UBER)

This module implements readers for keystores created by the BouncyCastle cryptographic provider for Java.

3.4.1 Store types

Pyjks supports two BouncyCastle store types:

- BKS
- UBER

Neither BKS or JKS/JCEKS stores make any effort to hide how many entries are present in the store, what their aliases are, and what type of key each entry contains. The keys *inside* each entry are still protected, and the store is protected against tampering via the store password, but anyone can still see the names and types of keys inside.

UBER keystores are similar to BKS, but they have an additional design goal: protect the store from introspection. This is done by additionally encrypting the entire keystore using (a key derived from) the store password.

class `jks.bks.`**`BksKeyStore`**(*store_type*, *entries*, *version=2*)

Bases: `jks.util.AbstractKeystore`

Bouncycastle "BKS" keystore parser. Supports both the current V2 and old V1 formats.

`entries`
A dictionary of all entries in the keystore, mapped by alias.

`store_type`
A string indicating the type of keystore that was loaded. Equals `bks` for instances of this class.

`classmethod load`(*filename*, *store_password*, *try_decrypt_keys=True*)
Convenience wrapper function; reads the contents of the given file and passes it through to *`loads()`*. See *`loads()`*.

`classmethod loads`(*data*, *store_password*, *try_decrypt_keys=True*)
See *`jks.jks.KeyStore.loads()`*.

Parameters

- **`data`** (*`bytes`*) – Byte string representation of the keystore to be loaded.
- **`password`** (*`str`*) – Keystore password string
- **`try_decrypt_keys`** (*`bool`*) – Whether to automatically try to decrypt any encountered key entries using the same password as the keystore password.

Returns

A loaded *`BksKeyStore`* instance, if the keystore could be successfully parsed and the supplied store password is correct.

If the `try_decrypt_keys` parameters was set to `True`, any keys that could be successfully decrypted using the store password have already been decrypted; otherwise, no atttempt to decrypt any key entries is made.

Raises

- ***`BadKeystoreFormatException`*** – If the keystore is malformed in some way
- ***`UnsupportedKeystoreVersionException`*** – If the keystore contains an unknown format version number
- ***`KeystoreSignatureException`*** – If the keystore signature could not be verified using the supplied store password
- ***`DuplicateAliasException`*** – If the keystore contains duplicate aliases

`save`(*filename*, *store_password*)
Convenience wrapper function; calls the `saves()` and saves the content to a file.

`certs`
A subset of the *`entries`* dictionary, filtered down to only those entries of type *`BksTrustedCertEntry`*.

`plain_keys`
A subset of the *`entries`* dictionary, filtered down to only those entries of type *`BksKeyEntry`*.

`sealed_keys`
A subset of the *`entries`* dictionary, filtered down to only those entries of type *`BksSealedKeyEntry`*.

`secret_keys`
A subset of the *`entries`* dictionary, filtered down to only those entries of type *`BksSecretKeyEntry`*.

version = None
Version of the keystore format, if loaded.

class jks.bks.**UberKeyStore**(*store_type*, *entries*, *version=1*)
Bases: `jks.bks.BksKeyStore`

BouncyCastle "UBER" keystore format parser.

entries
A dictionary of all entries in the keystore, mapped by alias.

store_type
A string indicating the type of keystore that was loaded. Equals `uber` for instances of this class.

classmethod load(*filename*, *store_password*, *try_decrypt_keys=True*)
Convenience wrapper function; reads the contents of the given file and passes it through to `loads()`. See `loads()`.

classmethod loads(*data*, *store_password*, *try_decrypt_keys=True*)
See `jks.jks.KeyStore.loads()`.

Parameters

- **data** (`bytes`) – Byte string representation of the keystore to be loaded.
- **password** (`str`) – Keystore password string
- **try_decrypt_keys** (`bool`) – Whether to automatically try to decrypt any encountered key entries using the same password as the keystore password.

Returns

A loaded `UberKeyStore` instance, if the keystore could be successfully parsed and the supplied store password is correct.

If the `try_decrypt_keys` parameters was set to `True`, any keys that could be successfully decrypted using the store password have already been decrypted; otherwise, no atttempt to decrypt any key entries is made.

Raises

- **BadKeystoreFormatException** – If the keystore is malformed in some way
- **UnsupportedKeystoreVersionException** – If the keystore contains an unknown format version number
- **KeystoreSignatureException** – If the keystore signature could not be verified using the supplied store password
- **DecryptionFailureException** – If the keystore contents could not be decrypted using the supplied store password
- **DuplicateAliasException** – If the keystore contains duplicate aliases

save(*filename*, *store_password*)
Convenience wrapper function; calls the `saves()` and saves the content to a file.

certs
A subset of the `entries` dictionary, filtered down to only those entries of type `BksTrustedCertEntry`.

plain_keys
A subset of the `entries` dictionary, filtered down to only those entries of type `BksKeyEntry`.

sealed_keys
A subset of the *entries* dictionary, filtered down to only those entries of type *BksSealedKeyEntry*.

secret_keys
A subset of the *entries* dictionary, filtered down to only those entries of type *BksSecretKeyEntry*.

version = None
Version of the keystore format, if loaded.

3.4.2 Entry types

`jks.bks.`**KEY_TYPE_PRIVATE = 0**
Type indicator for private keys in *BksKeyEntry*.

`jks.bks.`**KEY_TYPE_PUBLIC = 1**
Type indicator for public keys in *BksKeyEntry*.

`jks.bks.`**KEY_TYPE_SECRET = 2**
Type indicator for secret keys in *BksKeyEntry*. Indicates a key for use with a symmetric encryption algorithm.

class `jks.bks.`**BksTrustedCertEntry**(***kwargs*)
Bases: *jks.jks.TrustedCertEntry*

Represents a trusted certificate entry in a BKS or UBER keystore.

type
A string indicating the type of certificate. Unless in exotic applications, this is usually `X.509`.

cert
A byte string containing the actual certificate data. In the case of X.509 certificates, this is the DER-encoded X.509 representation of the certificate.

decrypt(*key_password*)
Does nothing for this entry type; certificates are inherently public data and are not stored in encrypted form.

encrypt(*key_password*)
Does nothing for this entry type; certificates are inherently public data and are not stored in encrypted form.

is_decrypted()
Always returns `True` for this entry type.

classmethod new(*alias*, *cert*)
Helper function to create a new TrustedCertEntry.

Parameters

- **alias** (*str*) – The alias for the Trusted Cert Entry
- **certs** (*str*) – The certificate, as a byte string.

Returns A loaded `TrustedCertEntry` instance, ready to be placed in a keystore.

class `jks.bks.`**BksKeyEntry**(*type*, *format*, *algorithm*, *encoded*, ***kwargs*)
Bases: `jks.bks.AbstractBksEntry`

Represents a non-encrypted cryptographic key (public, private or secret) stored in a BKS keystore. May exceptionally appear as a top-level entry type in (very) old keystores, but you are most likely to encounter these as the nested object inside a *BksSealedKeyEntry* once decrypted.

When *type* is `KEY_TYPE_PRIVATE`, the following attributes are available:

pkey

Note: Only accessible after a call to *decrypt()*; until then, accessing this attribute will raise a *NotYetDecryptedException*. See also `try_decrypt_keys` on *loads()*.

A byte string containing the value of the `privateKey` field of the PKCS#8 `PrivateKeyInfo` representation of the private key. See RFC 5208, section 5: Private-Key Information Syntax.

pkey_pkcs8

Note: Only accessible after a call to *decrypt()*; until then, accessing this attribute will raise a *NotYetDecryptedException*. See also `try_decrypt_keys` on *loads()*.

A byte string containing the DER-encoded PKCS#8 `PrivateKeyInfo` representation of the private key. See RFC 5208, section 5: Private-Key Information Syntax.

algorithm_oid

Note: Only accessible after a call to *decrypt()*; until then, accessing this attribute will raise a *NotYetDecryptedException*. See also `try_decrypt_keys` on *loads()*.

A tuple of integers corresponding to the algorithm OID for which the private key is valid.

Common values include:

- `(1,2,840,113549,1,1,1)` (alias `rsaEncryption`)
- `(1,2,840,10040,4,1)` (alias `id-dsa`).

When *type* is `KEY_TYPE_PUBLIC`, the following attributes are available:

public_key

Note: Only accessible after a call to *decrypt()*; until then, accessing this attribute will raise a *NotYetDecryptedException*. See also `try_decrypt_keys` on *loads()*.

A byte string containing the value of the `subjectPublicKey` field of the X.509 `SubjectPublicKeyInfo` representation of the public key. See RFC 5280, Appendix A. Pseudo-ASN.1 Structures and OIDs.

public_key_info

Note: Only accessible after a call to *decrypt()*; until then, accessing this attribute will raise a *NotYetDecryptedException*. See also `try_decrypt_keys` on *loads()*.

A byte string containing the DER-encoded X.509 `SubjectPublicKeyInfo` representation of the public key. See RFC 5280, Appendix A. Pseudo-ASN.1 Structures and OIDs.

algorithm_oid

> **Note:** Only accessible after a call to `decrypt()`; until then, accessing this attribute will raise a `NotYetDecryptedException`. See also `try_decrypt_keys` on `loads()`.

A tuple of integers corresponding to the algorithm OID for which the public key is valid.

Common values include:

- `(1,2,840,113549,1,1,1)` (alias `rsaEncryption`)
- `(1,2,840,10040,4,1)` (alias `id-dsa`).

When `type` is `KEY_TYPE_SECRET`, the following attributes are available:

key

> **Note:** Only accessible after a call to `decrypt()`; until then, accessing this attribute will raise a `NotYetDecryptedException`. See also `try_decrypt_keys` on `loads()`.

A byte string containing the raw secret key.

key_size

> **Note:** Only accessible after a call to `decrypt()`; until then, accessing this attribute will raise a `NotYetDecryptedException`. See also `try_decrypt_keys` on `loads()`.

An integer containing the size of the key, in bits. For DES and 3DES keys, the sizes 64 bits resp. 192 bits are returned.

type = None
An integer indicating the type of key: one of `KEY_TYPE_PRIVATE`, `KEY_TYPE_PUBLIC`, `KEY_TYPE_SECRET`.

format = None
A string indicating the format or encoding in which the key is stored. One of: `PKCS8`, `PKCS#8`, `X.509`, `X509`, `RAW`.

algorithm = None
A string indicating the algorithm for which the key is valid.

encoded = None
A byte string containing the key, formatted as indicated by the `format` attribute.

is_decrypted()
Always returns `True` for this entry type.

decrypt(*key_password*)
Does nothing for this entry type; these entries are stored in non-encrypted form.

classmethod type2str(*t*)
Returns a string representation of the given key type. Returns one of `PRIVATE`, `PUBLIC` or `SECRET`, or `None` if no such key type is known.

Parameters t (*int*) – Key type constant. One of `KEY_TYPE_PRIVATE`, `KEY_TYPE_PUBLIC`, `KEY_TYPE_SECRET`.

encrypt (*key_password*)
Encrypts the entry using the given password, so that it can be saved.

Parameters **key_password** (*str*) – The password to encrypt the entry with.

classmethod new (*alias*)
Helper function to create a new KeyStoreEntry.

class `jks.bks.`**BksSecretKeyEntry**(***kwargs*)
Bases: `jks.bks.AbstractBksEntry`

Conceptually similar to, but not to be confused with, *BksKeyEntry* objects of type `KEY_TYPE_SECRET`:

- *BksSecretKeyEntry* objects store the result of arbitrary user-supplied byte[]s, which, per the Java Keystore SPI, keystores are obligated to assume have already been protected by the user in some unspecified way. Because of this assumption, no password is provided for these entries when adding them to the keystore, and keystores are thus forced to store these bytes as-is.

  Produced by a call to `KeyStore.setKeyEntry(String alias, byte[] key, Certificate[] chain)` call.

  The bouncycastle project appears to have completely abandoned these entry types well over a decade ago now, and it is no longer possible to retrieve these entries through the Java APIs in any (remotely) recent BC version.

- *BksKeyEntry* objects of type `KEY_TYPE_SECRET` store the result of a getEncoded() call on proper Java objects of type SecretKey.

  Produced by a call to `KeyStore.setKeyEntry(String alias, Key key, char[] password, Certificate[] chain).`

  The difference here is that the KeyStore implementation knows it's getting a proper (Secret)Key Java object, and can decide for itself how to store it given the password supplied by the user. I.e., in this version of setKeyEntry it is left up to the keystore implementation to encode and protect the supplied Key object, instead of in advance by the user.

key = None
A byte string containing the secret key/value.

is_decrypted()
Always returns `True` for this entry type.

decrypt (*key_password*)
Does nothing for this entry type; these entries stored arbitrary user-supplied data, unclear how to decrypt (may not be encrypted at all).

encrypt (*key_password*)
Encrypts the entry using the given password, so that it can be saved.

Parameters **key_password** (*str*) – The password to encrypt the entry with.

classmethod new (*alias*)
Helper function to create a new KeyStoreEntry.

class `jks.bks.`**BksSealedKeyEntry**(***kwargs*)
Bases: `jks.bks.AbstractBksEntry`

PBEWithSHAAnd3-KeyTripleDES-CBC-encrypted wrapper around a *BksKeyEntry*. The contained key type is unknown until decrypted.

Once decrypted, objects of this type can be used in the same way as *BksKeyEntry*: attribute accesses are forwarded to the wrapped *BksKeyEntry* object.

is_decrypted()
Returns `True` if the entry has already been decrypted, `False` otherwise.

decrypt(*key_password*)
Decrypts the entry using the given password. Has no effect if the entry has already been decrypted.

Parameters **key_password** (*str*) – The password to decrypt the entry with.

Raises

- ***DecryptionFailureException*** – If the entry could not be decrypted using the given password.
- ***UnexpectedAlgorithmException*** – If the entry was encrypted with an unknown or unexpected algorithm

encrypt(*key_password*)
Encrypts the entry using the given password, so that it can be saved.

Parameters **key_password** (*str*) – The password to encrypt the entry with.

classmethod new(*alias*)
Helper function to create a new KeyStoreEntry.

3.5 Exceptions

All exceptions related to keystore loading or parsing derive from a common superclass type *KeystoreException*.

3.5.1 Exception types

exception jks.util.**KeystoreException**
Bases: `exceptions.Exception`

Superclass for all pyjks exceptions.

exception jks.util.**KeystoreSignatureException**
Bases: *jks.util.KeystoreException*

Signifies that the supplied password for a keystore integrity check is incorrect.

exception jks.util.**DuplicateAliasException**
Bases: *jks.util.KeystoreException*

Signifies that duplicate aliases were encountered in a keystore.

exception jks.util.**NotYetDecryptedException**
Bases: *jks.util.KeystoreException*

Signifies that an attribute of a key store entry can not be accessed because the entry has not yet been decrypted.

By default, the keystore `load` and `loads` methods automatically try to decrypt all key entries using the store password. Any keys for which that attempt fails are returned undecrypted, and will raise this exception when its attributes are accessed.

To resolve, first call decrypt() with the correct password on the entry object whose attributes you want to access.

exception jks.util.**BadKeystoreFormatException**
Bases: *jks.util.KeystoreException*

Signifies that a structural error was encountered during key store parsing.

exception `jks.util.`**`BadDataLengthException`**
Bases: *`jks.util.KeystoreException`*

Signifies that given input data was of wrong or unexpected length.

exception `jks.util.`**`BadPaddingException`**
Bases: *`jks.util.KeystoreException`*

Signifies that bad padding was encountered during decryption.

exception `jks.util.`**`BadHashCheckException`**
Bases: *`jks.util.KeystoreException`*

Signifies that a hash computation did not match an expected value.

exception `jks.util.`**`DecryptionFailureException`**
Bases: *`jks.util.KeystoreException`*

Signifies failure to decrypt a value.

exception `jks.util.`**`UnsupportedKeystoreVersionException`**
Bases: *`jks.util.KeystoreException`*

Signifies an unexpected or unsupported keystore format version.

exception `jks.util.`**`UnexpectedJavaTypeException`**
Bases: *`jks.util.KeystoreException`*

Signifies that a serialized Java object of unexpected type was encountered.

exception `jks.util.`**`UnexpectedAlgorithmException`**
Bases: *`jks.util.KeystoreException`*

Signifies that an unexpected cryptographic algorithm was used in a keystore.

exception `jks.util.`**`UnexpectedKeyEncodingException`**
Bases: *`jks.util.KeystoreException`*

Signifies that a key was stored in an unexpected format or encoding.

exception `jks.util.`**`UnsupportedKeystoreTypeException`**
Bases: *`jks.util.KeystoreException`*

Signifies that the keystore was an unsupported type.

exception `jks.util.`**`UnsupportedKeystoreEntryTypeException`**
Bases: *`jks.util.KeystoreException`*

Signifies that the keystore entry was an unsupported type.

exception `jks.util.`**`UnsupportedKeyFormatException`**
Bases: *`jks.util.KeystoreException`*

Signifies that the key format was an unsupported type.

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index