Python PKCS#11 Documentation

Danielle Madeley

Feb 22, 2021

CONTENTS

A high level, "more Pythonic" interface to the PKCS#11 (Cryptoki) standard to support HSM and Smartcard devices in Python.

The interface is designed to follow the logical structure of a HSM, with useful defaults for obscurely documented parameters. Many APIs will optionally accept iterables and act as generators, allowing you to stream large data blocks for symmetric encryption.

python-pkcs11 also includes numerous utility functions to convert between PKCS #11 data structures and common interchange formats including PKCS #1 and X.509.

python-pkcs11 is fully documented and has a full integration test suite for all features, with continuous integration against multiple HSM platforms including:

- Thales nCipher
- Opencryptoki TPM
- OpenSC/Smartcard-HSM/Nitrokey HSM

Source: https://github.com/danni/python-pkcs11

Documentation: http://python-pkcs11.readthedocs.io/en/latest/

CHAPTER ONE

GETTING STARTED

Install from Pip:

```
pip install python-pkcs11
```

Or build from source:

```
python setup.py build
```

Assuming your PKCS#11 library is set as *PKCS11_MODULE* and contains a token named *DEMO*:

1.1 AES

```
import pkcs11

# Initialise our PKCS#11 library
lib = pkcs11.lib(os.environ['PKCS11_MODULE'])
token = lib.get_token(token_label='DEMO')

data = b'INPUT DATA'

# Open a session on our token
with token.open(user_pin='1234') as session:
    # Generate an AES key in this session
    key = session.generate_key(pkcs11.KeyType.AES, 256)

    # Get an initialisation vector
    iv = session.generate_random(128)  # AES blocks are fixed at 128 bits
    # Encrypt our data
    crypttext = key.encrypt(data, mechanism_param=iv)
```

1.2 3DES

```
import pkcs11

# Initialise our PKCS#11 library
lib = pkcs11.lib(os.environ['PKCS11_MODULE'])
token = lib.get_token(token_label='DEMO')

```

(continues on next page)

(continued from previous page)

```
data = b'INPUT DATA'

# Open a session on our token
with token.open(user_pin='1234') as session:
    # Generate a DES key in this session
    key = session.generate_key(pkcs11.KeyType.DES3)

    # Get an initialisation vector
    iv = session.generate_random(64)  # DES blocks are fixed at 64 bits
    # Encrypt our data
    crypttext = key.encrypt(data, mechanism_param=iv)
```

1.3 RSA

```
import pkcs11

lib = pkcs11.lib(os.environ['PKCS11_MODULE'])
token = lib.get_token(token_label='DEMO')

data = b'INPUT DATA'

# Open a session on our token
with token.open(user_pin='1234') as session:
    # Generate an RSA keypair in this session
    pub, priv = session.generate_keypair(pkcs11.KeyType.RSA, 2048)

    # Encrypt as one block
    crypttext = pub.encrypt(data)
```

1.4 DSA

```
import pkcs11

lib = pkcs11.lib(os.environ['PKCS11_MODULE'])
token = lib.get_token(token_label='DEMO')

data = b'INPUT DATA'

# Open a session on our token
with token.open(user_pin='1234') as session:
    # Generate an DSA keypair in this session
    pub, priv = session.generate_keypair(pkcs11.KeyType.DSA, 1024)

    # Sign
    signature = priv.sign(data)
```

1.5 ECDSA

```
import pkcs11

lib = pkcs11.lib(os.environ['PKCS11_MODULE'])
token = lib.get_token(token_label='DEMO')

data = b'INPUT DATA'

# Open a session on our token
with token.open(user_pin='1234') as session:
    # Generate an EC keypair in this session from a named curve
    ecparams = session.create_domain_parameters(
        pkcs11.KeyType.EC, {
            pkcs11.Attribute.EC_PARAMS: pkcs11.util.ec.encode_named_curve_parameters(
↪'secp256r1'),
        }, local=True)
    pub, priv = ecparams.generate_keypair()

    # Sign
    signature = priv.sign(data)
```

1.6 Diffie-Hellman

```
import pkcs11

lib = pkcs11.lib(os.environ['PKCS11_MODULE'])
token = lib.get_token(token_label='DEMO')

with token.open() as session:
    # Given shared Diffie-Hellman parameters
    parameters = session.create_domain_parameters(pkcs11.KeyType.DH, {
        pkcs11.Attribute.PRIME: prime,  # Diffie-Hellman parameters
        pkcs11.Attribute.BASE: base,
    })

    # Generate a DH key pair from the public parameters
    public, private = parameters.generate_keypair()

    # Share the public half of it with our other party.
    _network_.write(public[Attribute.VALUE])
    # And get their shared value
    other_value = _network_.read()

    # Derive a shared session key with perfect forward secrecy
    session_key = private.derive_key(
        pkcs11.KeyType.AES, 128,
        mechanism_param=other_value)
```

1.7 Elliptic-Curve Diffie-Hellman

```
import pkcs11

lib = pkcs11.lib(os.environ['PKCS11_MODULE'])
token = lib.get_token(token_label='DEMO')

with token.open() as session:
    # Given DER encocded EC parameters, e.g. from
    #    openssl ecparam -outform der -name <named curve>
    parameters = session.create_domain_parameters(pkcs11.KeyType.EC, {
        pkcs11.Attribute.EC_PARAMS: ecparams,
    })

    # Generate a DH key pair from the public parameters
    public, private = parameters.generate_keypair()

    # Share the public half of it with our other party.
    _network_.write(public[pkcs11.Attribute.EC_POINT])
    # And get their shared value
    other_value = _network_.read()

    # Derive a shared session key
    session_key = private.derive_key(
        pkcs11.KeyType.AES, 128,
        mechanism_param=(pkcs11.KDF.NULL, None, other_value))
```

CHAPTER TWO

TESTED COMPATIBILITY

Functionality		SoftHSMv2	Thales nCipher	Opencryptoki	OpenSC (Nitrokey)
Get Slots/Tokens		Works	Works	Works	Works
Get Mechanisms		Works	Works	Works	Works
Initialize token		Not implemented			
Slot events		Not implemented			
Alternative authentication path		Not implemented			
Always authenticate keys		Not implemented			
Create/Copy	Keys	Works	Works	Errors	Create
	Certificates	Caveats[1]	Caveats[1]	Caveats[1]	?
	Domain Params	Caveats[1]	Caveats[1]	?	N/A
Destroy Object		Works	N/A	Works	Works
Generate Random		Works	Works	Works	Works
Seed Random		Works	N/A	N/A	N/A
Digest (Data & Keys)		Works	Caveats[2]	Works	Works
AES	Generate key	Works	Works	Works	N/A
	Encrypt/Decrypt	Works	Works	Works	
	Wrap/Unwrap	?[3]	Works	Errors	
	Sign/Verify	Works	Works[4]	N/A	
DES2/ DES3	Generate key	Works	Works	Works	N/A
	Encrypt/Decrypt	Works	Works	Works	
	Wrap/Unwrap	?	?	?	
	Sign/Verify	?	?	?	
RSA	Generate key pair	Works	Works	Works	Works[4][8]
	Encrypt/Decrypt	Works	Works	Works	Decrypt only[9]
	Wrap/Unwrap	Works	Works	Works	N/A
	Sign/Verify	Works	Works	Works	Works
DSA	Generate parameters	Works	Error	N/A	N/A
	Generate key pair	Works	Caveats[5]		
	Sign/Verify	Works	Works[4]		
DH	Generate parameters	Works	N/A	N/A	N/A
	Generate key pair	Works	Caveats[6]		
	Derive Key	Works	Caveats[7]		
EC	Generate key pair	Caveats[6]	?[3]	N/A	Works
	Sign/Verify (ECDSA)	Works[4]	?[3]		Sign only[9]
	Derive key (ECDH)	Works	?[3]		?

Continued on next page

Table 1 – continued from previous page

Functionality	SoftHSMv2	Thales nCipher	Opencryptoki	OpenSC (Nitrokey)
Proprietary extensions	N/A	Not implemented	N/A	N/A

Python version:

- 3.4 (with *aenum*)
- 3.5 (with *aenum*)
- 3.6

PKCS#11 versions:

- 2.11
- 2.20
- 2.40

Feel free to send pull requests for any functionality that's not exposed. The code is designed to be readable and expose the PKCS #11 spec in a straight-forward way.

If you want your device supported, get in touch!

[1] Device supports limited set of attributes.
[2] Digesting keys is not supported.
[3] Untested: requires support in device.
[4] Default mechanism not supported, must specify a mechanism.
[8] *store* parameter is ignored, all keys are stored.
[9] Encryption/verify not supported, extract the public key
[5] From existing domain parameters.
[6] Local domain parameters only.
[7] Generates security warnings about the derived key.

CHAPTER

THREE

MORE INFO ON PKCS #11

The latest version of the PKCS #11 spec is available from OASIS:

http://docs.oasis-open.org/pkcs11/pkcs11-base/v2.40/pkcs11-base-v2.40.html

You should also consult the documentation for your PKCS #11 implementation. Many implementations expose additional vendor options configurable in your environment, including alternative features, modes and debugging information.

CHAPTER FOUR

LICENSE

MIT License

CHAPTER

FIVE

CONTENTS

5.1 Applied PKCS #11

PKCS #11 is the name given to a standard defining an API for cryptographic hardware. While it was developed by RSA, as part of a suite of standards, the standard is not exclusive to RSA ciphers and is meant to cover a wide range of cryptographic possibilities. PKCS #11 is most closely related to Java's JCE and Microsoft's CAPI.

Section Contents

5.1.1 Concepts in PKCS #11

Slots and Tokens

A *slot* originally referred to a single card slot on a smartcard device that could accept a *token*. A token was a smartcard that contained secure, encrypted keys and certificates. You would insert your smartcard (token) into the slot, and use its contents to do cryptographic operations.

Nowadays the distinction is more blurry. Many USB-key HSMs appear as a single slot containing a hardwired single token (their internal storage). Server devices often make use of software tokens (*softcards*), which appear as slots within PKCS #11, but no physical device exists. These devices can also feature physical slots and *accelerator slots*.

See also:

Slots have `pkcs11.Slot.flags` which can tell you something about what kind of slot this is.

Tokens are secured with a passphrase (PIN). Not all implementations use pins in their underlying implementation, but these are required for PKCS#11. Some implementations let you control the behaviour of their PKCS #11 module in ways not specified by the specification through environment variables (e.g. default token pins).

Note: The PKCS #11 library is running within your process, using your memory, etc. It may talk to a daemon to access the underlying hardware, or it may be talking directly.

Environment variables set on your process can be used to configure the behaviour of the library, check the documentation for your device.

Finding Tokens

Tokens are identified by a label or serial number.

You can retrieve all tokens matching search parameters:

```
for slot in lib.get_slots():
    token = slot.get_token()
    # Check the parameters
    if token.label == '...':
        break
```

```
for token in lib.get_tokens(token_label='smartcard'):
    print(token)
```

Retrieving a single token has a shortcut function:

```
try:
    lib.get_token(token_label='smartcard')
except NoSuchToken:
    pass
except MultipleTokensReturned:
    pass
```

Mechanisms and Capabilities

Different devices support different cryptographic operations. In PKCS #11 mechanisms refer to the combination of cipher (e.g. AES), hash function (e.g. SHA512) and block mode (e.g. CBC). Mechanisms also exist for generating keys, and deriving keys and parameters.

The capabilities of a mechanism indicate what types of operations can be carried out with the mechanism, e.g. encryption, signing, key generation.

Not all devices support all mechanisms. Some may support non-standard mechanisms. Not all devices support the same capabilities for mechanisms or same key lengths. This information can be retrieved via `pkcs11.Slot.get_mechanisms()` and `pkcs11.Slot.get_mechanism_info()` or from your device documentation.

Some mechanisms require *mechanism parameters*. These are used to provide additional context to the mechanism that does not form part of the key. Examples of mechanism parameters are initialisation vectors for block modes, salts, key derivation functions, and other party's shared secrets (for Diffie-Hellman).

See also:

The `pkcs11.mechanisms.Mechanism` type includes information on the required parameters for common mechanisms. A complete list of current mechanisms and historical mechanisms includes the mechanism parameters and input requirements for each mechanism.

Objects and Attributes

An object is a piece of cryptographic information stored on a *token*. Objects have a *class* (e.g. private key) which is exposed in *python-pkcs11* as a Python class. They also have a number of other attributes depending on their class.

There are three main classes of object:

- keys (symmetric secret keys and asymmetric public and private keys);
- domain parameters (storing the parameters used to generate keys); and
- certificates (e.g. X.509 certificates).

Note: Irregardless of the PKCS #11 specification, not all devices reliably handle all object attributes. They can also have different defaults. *python-pkcs11* tries to abstract that as much as possible to enable writing portable code.

See also:

`pkcs11.constants.Attribute` describes the available attributes and their Python types.

biginteger

One type is handled specially: *biginteger*, an arbitrarily long integer in network byte order. Although Python can handle arbitrarily long integers, many other systems cannot and pass these types around as byte arrays, and more often than not, that is an easier form to handle them in.

biginteger attributes can be specified as `bytes`, `bytearray` or an iterable of byte-sized integers.

If you do have integers, you can convert them to `bytes` using `pkcs11.util.biginteger()`.

Finding Objects

Objects can be found on a *token* using their attributes. Usually an *ID* or *LABEL*.

```
for obj in session.get_objects({
    Attribute.CLASS: ObjectClass.SECRET_KEY,
    Attribute.LABEL: 'aes256',
}):
    print(obj)
```

Finding a specific key is so common there's a shortcut function:

```
try:
    key = session.get_key(label='aes256')
except NoSuchKey:
    pass
except MultipleObjectsReturned:
    pass
```

Keys

There are three classes of key objects:

- symmetric secret keys;
- asymmetric public keys; and
- asymmetric private keys.

The following attributes can be set for keys:

PRIVATE Private objects can only be accessed by logged in sessions.

LOCAL This key was generated on the device.

EXTRACTABLE The key can be extracted from the HSM.

SENSITIVE The key is sensitive and cannot be removed from the device in clear text.

ALWAYS_SENSITIVE The key has never not been *SENSITIVE*.

NEVER_EXTRACTABLE The key has never been *EXTRACTABLE*.

ALWAYS_AUTHENTICATE The key requires authentication every time it's used.

Note: Keys should be generated on the HSM rather than imported. Generally only public keys should not be *PRIVATE* and *SENSITIVE*. Allowing private keys to be accessed defeats the purpose of securing your keys in a HSM. *python-pkcs11* sets meaningful defaults.

Domain Parameters

Domain parameters are the parameters used to generate cryptographic keys (e.g. the name of the elliptic curve being used). They are public information. Obscuring the domain parameters does not increase the security of a cryptosystem. Typically the domain parameters form part of a protocol specification, and RFCs exist giving pre-agreed, named domain parameters for cryptosystems.

In *python-pkcs11* domain parameters can either be stored as an object in your HSM, or loaded via some other mechanism (e.g. in your code) and used directly without creating a HSM object.

See also:

OpenSSL can be used to generate unique or named domain parameters for Diffie-Hellman, DSA and EC.

`pkcs11.util` includes modules for creating and decoding domain parameters.

Sessions

Accessing a token is done by opening a session. Sessions can be public or logged in. Only a logged in session can access objects marked as *private*. Depending on your device, some functions may also be unavailable.

Warning: It is important to close sessions when you are finished with them. Some devices will leak resources if sessions aren't closed.

Where possible you should use sessions via a context manager.

5.1.2 Concepts related to PKCS #11

Binary Formats and Padding

PKCS #11 is *protocol agnostic* and does not define or implement any codecs for the storing of enciphered data, keys, initialisation vectors, etc. outside the HSM.[1] For example, CBC mechanisms will not include the initialization vector. You must choose a storage/transmission format that suits your requirements.

Some mechanisms require input data to be *padded* to a certain block size. Standardized *PAD* variants of many mechanisms exist based on upstream specifications. For other mechanisms PKCS #11 does not define any specific algorithms, and you must choose one that suits your requirements.

See also:

Lots of standards exist for the storing and transmission of cryptographic data. If you're not implementing a specific protocol, there may still be an RFC standard with a Python implementation to ensure people can understand your binary data in the future.

See also:

- RFC 5652 (Cryptographic Message Standard) (supercedes PKCS #7)

PKCS #15

PKCS #15 defines a standard for storing cryptographic objects within the HSM device to enable interoperability between devices and tokens. PKCS #15 is often referenced in conjunction with PKCS #11 as the storage format used on the *tokens*.

ASN.1, DER, BER

ASN.1 is a data model for storing structured information. DER and BER are binary representations of that data model which are used extensively in cryptography, e.g. for storing RSA key objects, X.509 certificates and elliptic curve information.

Accessing ASN.1 encoded objects is mostly left to packages other than *python-pkcs11*, however `pkcs11.util` does include some utilities to encode and decode objects where required for working with PKCS #11 itself (e.g. converting PKCS #1 encoded RSA keys into PKCS #11 objects and generating parameters for elliptic curves).

PEM

PEM is a standard for handling cryptographic objects. It is a base64 encoded version of the binary DER object. The label indicates the type of object, and thus what ASN.1 model to use. *python-pkcs11* does not include PEM parsing, you should include another package if required. `asn1crypto.pem` is a dependency of *python-pkcs11*.

[1] It does define types for data *inside* the HSM, e.g. attribute data types and binary formats (e.g. EC parameters, X.509 certificates).

5.1.3 Getting a Session

Given a PKCS #11 library (*.so*) that is stored in the environment as *PKCS11_MODULE*.

To open a read-only session on a token named *smartcard*:

```
import pkcs11

lib = pkcs11.lib(os.environ['PKCS11_MODULE'])
token = lib.get_token(token_label='smartcard')

with token.open() as session:
    print(session)
```

To open a user session with the passphrase/pin *secret*:

```
with token.open(user_pin='secret') as session:
    print(session)
```

To open a read/write session:

```
with token.open(rw=True, user_pin='secret') as session:
    print(session)
```

See also:

`pkcs11.Token.open()` has more options for opening the session.

5.1.4 Generating Keys

Keys can either live for the lifetime of the *session* or be stored on the token. Storing keys requires a read only session.

To store keys pass *store=True*. When storing keys it is recommended to set a *label* or *id*, so you can find the key again.

Symmetric Keys

AES

AES keys can be generated by specifying the key length:

```
from pkcs11 import KeyType

key = session.generate_key(KeyType.AES, 256)
```

Generally AES keys are considered secret. However if you're using your HSM to generate keys for use with local AES (e.g. in hybrid encryption systems). You can do the following:

```
from pkcs11 import KeyType, Attribute

key = session.generate_key(KeyType.AES, 256, template={
    Attribute.SENSITIVE: False,
    Attribute.EXTRACTABLE: True,
})
# This is the secret key
print(key[Attribute.VALUE])
```

VALUE Secret key (as *biginteger*).

DES2/3

> **Warning:** DES2 and DES3 are considered insecure because their short key lengths are brute forcable with modern hardware.

DES2/3 keys are fixed length.

```
from pkcs11 import KeyType

des2 = session.generate_key(KeyType.DES2)
des3 = session.generate_key(KeyType.DES3)
```

These secret key objects have the same parameters as for AES.

Asymmetric Keypairs

RSA

RSA keypairs can be generated by specifying the length of the modulus:

```
from pkcs11 import KeyType

public, private = session.generate_keypair(KeyType.RSA, 2048)
```

The default public exponent is *65537*. You can specify an alternative:

```
from pkcs11 import KeyType, Attribute

public, private = session.generate_keypair(KeyType.RSA, 2048,
                                           public_template={Attribute.PUBLIC_
↪EXPONENT: ...})
# This is the public key
print(public[Attribute.MODULUS])
print(public[Attribute.PUBLIC_EXPONENT])
```

The public key has two parameters:

MODULUS Key modulus (as *biginteger*).

PUBLIC_EXPONENT Public exponent (as *biginteger*).

These can be exported as RFC 2437 (PKCS #1) DER-encoded binary using *pkcs11.util.rsa.encode_rsa_public_key()*.

DSA

DSA keypairs can be generated by specifying the length of the prime in bits.

```
from pkcs11 import KeyType

public, private = session.generate_keypair(KeyType.RSA, 2048)
```

This will generate unique domain parameters for a key. If you want to create a key for given domain parameters, see *DSA from Domain Parameters*.

The public key has a single important attribute:

VALUE Public key (as biginteger).

This can be encoded in RFC 3279 format with `pkcs11.util.dsa.encode_dsa_public_key()`.

From Domain Parameters

Note: Choosing domain parameters is not covered in this document. Domain parameters are often either specified by the requirements you are implementing for, or have a standard implementation to derive quality parameters. Some domain parameters (e.g. choice of elliptic curve) can drastically weaken the cryptosystem.

DSA

Diffie-Hellman key pairs require three domain parameters, specified as *bigintegers*.

BASE The prime base (g) (as *biginteger*).

PRIME The prime modulus (p) (as *biginteger*).

SUBPRIME The subprime (q) (as *biginteger*).

```
from pkcs11 import Attribute

parameters = session.create_domain_parameters(KeyType.DSA, {
    Attribute.PRIME: b'prime...',
    Attribute.BASE: b'base...',
    Attribute.SUBPRIME: b'subprime...',
}, local=True)

public, private = parameters.generate_keypair()
```

RFC 3279 defines a standard ASN.1 encoding for DSA parameters, which can be loaded with `pkcs11.util.dsa.decode_dsa_domain_parameters()`:

```
params = session.create_domain_parameters(
    KeyType.DSA,
    decode_dsa_domain_parameters(b'DER-encoded parameters'),
    local=True)
```

If supported, unique domain parameters can also be generated for a given *PRIME* length (e.g. 1024 bits) with `pkcs11.Session.generate_domain_parameters()`:

```
params = session.generate_domain_parameters(KeyType.DSA, 1024)
```

These can be encoded into the standard ASN.1 DER encoding using *pkcs11.util.dsa.encode_dsa_domain_parameters()*.

Note: You can create a DSA key directly from freshly generated domain parameters with `Session.generate_keypair()`.

Diffie-Hellman

Diffie-Hellman key pairs require several domain parameters, specified as *bigintegers*. There are two forms of Diffie-Hellman domain parameters: PKCS #3 and X9.42.

BASE The prime base (g) (as *biginteger*).

PRIME The prime modulus (p) (as *biginteger*).

SUBPRIME (X9.42 only) The subprime (q) (as *biginteger*).

```
from pkcs11 import Attribute

parameters = session.create_domain_parameters(KeyType.DH, {
    Attribute.PRIME: b'prime...',
    Attribute.BASE: b'base...',
}, local=True)

public, private = parameters.generate_keypair()
```

RFC 3279 defines a standard ASN.1 encoding for DH parameters, which can be loaded with `pkcs11.util.dh.decode_x9_42_dh_domain_parameters()`:

```
params = session.create_domain_parameters(
    KeyType.X9_42_DH,
    decode_x9_42_dh_domain_parameters(b'DER-encoded parameters'),
    local=True)
```

If supported, unique domain parameters can also be generated for a given *PRIME* length (e.g. 512 bits) with *pkcs11.Session.generate_domain_parameters()*:

```
params = session.generate_domain_parameters(KeyType.DH, 512)
```

X9.42 format domain parameters can be encoded back to their RFC 3279 format with `pkcs11.util.dh.encode_x9_42_dh_domain_parameters()`.

Key pairs can be generated from the domain parameters:

```
public, private = parameters.generate_keypair()
# This is the public key
print(public[Attribute.VALUE])
```

The public key has a single important attribute:

VALUE Public key (as biginteger).

This can be encoded in RFC 3279 format with *pkcs11.util.dh.encode_dh_public_key()*.

Elliptic Curve

Elliptic curves require a domain parameter describing the curve. Curves can be described in two ways:

- As named curves; or
- As a complete set of parameters.

Not all devices support both specifications. You can determine what curve parameters your device supports by checking *pkcs11.Slot.get_mechanism_info()* *pkcs11.constants.MechanismFlag*.

Both specifications are specified using the same *attribute*:

EC_PARAMS Curve parameters (as DER-encoded X9.62 bytes).

```
from pkcs11 import Attribute


parameters = session.create_domain_parameters(KeyType.EC,
    Attribute.EC_PARAMS: b'DER-encoded X9.62 parameters ...',
}, local=True)

public, private = parameters.generate_keypair()
```

Named curves (e.g. *secp256r1*) can be specified like this:

```
from pkcs11 import Attribute
from pkcs11.util.ec import encode_named_curve_parameters


parameters = session.create_domain_parameters(KeyType.EC, {
    Attribute.EC_PARAMS: encode_named_curve_parameters('secp256r1')
}, local=True)
```

Key pairs can be generated from the domain parameters:

```
public, private = parameters.generate_keypair()
# This is the public key
print(public[Attribute.EC_POINT])
```

The public key as a single important attribute:

EC_POINT Public key (as X9.62 DER-encoded bytes).

5.1.5 Importing/Exporting Keys

Warning: It is best to only import/export public keys. You should, whenever possible, generate and store secret and private keys within the boundary of your HSM.

The following utility methods will convert keys encoded in their canonical DER-encoded into attributes that can be used with *pkcs11.Session.create_object()*.

Note: PEM certificates are base64-encoded versions of the canonical DER-encoded forms used in *python-pkcs11*. Conversion between PEM and DER can be achieved using asn1crypto.pem.

AES/DES

Warning: Whenever possible, generate and store secret keys within the boundary of your HSM.

AES and DES keys are stored as binary bytes in *pkcs11.constants.Attribute.VALUE*.

Keys must be marked as *EXTRACTABLE* and not *SENSITIVE* to export.

RSA

To import a PKCS #1 DER-encoded RSA key, the following utility methods are provided:

- *pkcs11.util.rsa.decode_rsa_public_key()*, and
- *pkcs11.util.rsa.decode_rsa_private_key()*.

To export an RSA public key in PKCS #1 DER-encoded format, use *pkcs11.util.rsa.encode_rsa_public_key()*.

DSA

To import an RFC 3279 DER-encoded DSA key, the following utility methods are provided:

- *pkcs11.util.dsa.decode_dsa_domain_parameters()*, and
- *pkcs11.util.dsa.decode_dsa_public_key()*.

To export a DSA public key, use:

- *pkcs11.util.dsa.encode_dsa_domain_parameters()*, and
- *pkcs11.util.dsa.encode_dsa_public_key()*.

Elliptic Curve

The *pkcs11.constants.Attribute.EC_PARAMS* and *pkcs11.constants.Attribute.EC_POINT* attributes for elliptic curves are already in DER-encoded X9.62 format.

You can import keys from OpenSSL using:

- *pkcs11.util.ec.decode_ec_public_key()*, and
- *pkcs11.util.ec.decode_ec_private_key()*.

To export an EC public key in OpenSSL format, use *pkcs11.util.ec.encode_ec_public_key()*.

X.509

The function *pkcs11.util.x509.decode_x509_public_key()* is provided to extract public keys from X.509 DER-encoded certificates, which is capable of handling RSA, DSA and ECDSA keys.

5.1.6 Encryption/Decryption

Ciphers can generally be considered in two categories:

- Symmetric ciphers (e.g. AES), which use a single key to encrypt and decrypt, and are good at encrypting large amounts of data; and
- Asymmetric ciphers (e.g. RSA), which use separate public and private keys, and are good for securing small amounts of data.

Symmetric ciphers operate on blocks of data, and thus are used along with a block mode. *python-pkcs11* can consume block mode ciphers via a generator.

Asymmetric ciphers are used for public-key cryptography. They cannot encrypt large amounts of data. Typically these ciphers are used to encrypt a symmetric session key, which does the bulk of the work, in a so-called hybrid cryptosystem.

Cipher	Block modes	Block Size (IV len)	Mechanism Param
AES	Yes	128 bits	IV (except EBC)
DES2/3	Yes	64 bits	IV (except EBC)
RSA	No	N/A	Optional

AES

The AES cipher requires you to specify a block mode as part of the *mechanism*.

The default block mode is CBC with PKCS padding, which can handle data not padded to the block size and requires you to supply an initialisation vector of 128-bits of good random.

A number of other mechanisms are available:

Mechanism	IV	Input Size	Notes
AES_ECB	No	128-bit blocks	Only suitable for key-wrapping. Identical blocks encrypt identically!
AES_CBC	Yes	128-bit blocks	
AES_CBC_PAD	Yes	Any	Default mechanism
AES_OFB	Yes	Any	
AES_CFB_*	Yes	Any	3 modes: AES_CFB8, AES_CFB64, and AES_CFB128.
AES_CTS	Yes	>= 128-bit	
AES_CTR	Not currently supported[2]		
AES_GCM			
AES_CGM			

[2] AES encryption with multiple mechanism parameters not currently implemented due to lack of hardware supporting these mechanisms.

Warning: Initialisation vectors

An initialization vector (IV) or starting variable (SV) is data that is used by several modes to randomize the encryption and hence to produce distinct ciphertexts even if the same plaintext is encrypted multiple times.

An initialization vector has different security requirements than a key, so the IV usually does not need to be secret. However, in most cases, it is important that an initialization vector is never reused under the same key. For CBC and CFB, reusing an IV leaks some information about the first block of plaintext, and about any common prefix shared by the two messages. For OFB and CTR, reusing an IV completely destroys security.

In CBC mode, the IV must, in addition, be unpredictable at encryption time; in particular, the (previously) common practice of re-using the last ciphertext block of a message as the IV for the next message is insecure.

We recommend using `pkcs11.Session.generate_random()` to create a quality IV.

A simple example:

```
# Given an AES key `key`
iv = session.generate_random(128)
ciphertext = key.encrypt(plaintext, mechanism_param=iv)

plaintext = key.decrypt(ciphertext, mechanism_param=iv)
```

Or using an alternative mechanism:

```
from pkcs11 import Mechanism

iv = session.generate_random(128)
ciphertext = key.encrypt(plaintext,
                         mechanism=Mechanism.AES_OFB,
                         mechanism_param=iv)
```

Large amounts of data can be passed as a generator:

```
buffer_size = 8192
with \\
        open(file_in, 'rb') as input, \\
        open(file_out, 'wb') as output:

    # A generator yielding chunks of the file
    chunks = iter(lambda: input.read(buffer_size), '')

    for chunk in key.encrypt(chunks,
                             mechanism_param=iv,
                             buffer_size=buffer_size):
        output.write(chunk)
```

Note: These mechanisms do not store the IV. You must store the IV yourself, e.g. on the front of the ciphertext. It is safe to store an IV in the clear.

DES2/3

> **Warning:** DES2 and DES3 are considered insecure because their short key lengths are brute forcable with modern hardware.

DES2/3 have the same block mode options as AES. The block size is 64 bits, which is the size of the initialization vector.

```
# Given an DES3 key `key`
iv = session.generate_random(64)
ciphertext = key.encrypt(plaintext, mechanism_param=iv)

plaintext = key.decrypt(ciphertext, mechanism_param=iv)
```

RSA

The default RSA cipher is PKCS #1 OAEP

A number of other mechanisms are available:

Mechanism	Parameters	Input Length	Notes
RSA_PKCS	None	<= key length - 11	RSA v1.5. Don't use for new applications.
RSA_PKCS_OAEP	See below	<= k - 2 - 2hLen	Default mechanism.
RSA_X_509	None	key length	Raw mode. No padding.
RSA_PKCS_TPM_1_1	None	<= key length - 11 - 5	See TCPA TPM Specification Version 1.1b
RSA_PKCS_OAEP_TPM_1_1	See below	<= k - 2 - 2hLen	

A simple example using the default parameters:

```
# Given an RSA key pair `public, private`
ciphertext = public.encrypt(plaintext)

plaintext = private.decrypt(ciphertext)
```

RSA OAEP can optionally take a tuple of *(hash algorithm, mask generating function and source data)* as the mechanism parameter:

```
ciphertext = public.encrypt(plaintext,
                            mechanism=Mechanism.RSA_PKCS_OAEP,
                            mechanism_param=(Mechanism.SHA_1,
                                             MGF.SHA1,
                                             None))
```

5.1.7 Signing/Verifying

Signing and verification mechanisms require two components:

- the cipher; and
- the hashing function.

Raw versions for some mechanisms also exist. These require you to do your own hashing outside of PKCS #11.

Signing functions typically work on a finite length of data, so the signing of large amounts of data requires hashing with a secure one-way hash function.

AES

A *MAC* is required for signing with AES. The default mechanism is *AES_MAC*.

```
# Given a secret key, `key`
signature = key.sign(data)

assert key.verify(data, signature)
```

DES2/3

A *MAC* is required for signing with DES. The default mechanism is *SHA512_HMAC* (aka HMAC-SHA512).

Operation is the same as for *AES*.

RSA

The default signing and verification mechanism for RSA is *RSA_SHA512_PKCS*.

Other mechanisms are available:

Mechanism	Notes
RSA_PKCS	No hashing. Supply your own.
SHA*_RSA_PKCS	SHAx message digesting.
RSA_PKCS_PSS	Optionally takes a tuple of parameters.
SHA*_RSA_PKCS_PSS	
RSA_9796	ISO/IES 9796 RSA signing. Use *PSS* instead.
RSA_X_509	X.509 (raw) RSA signing. You must supply your own padding.
RSA_X9_31	X9.31 RSA signing.

Simple example using the default mechanism:

```
# Given a private key `private`
signature = private.sign(data)

# Given a public key `public`
assert public.verify(data, signature)
```

RSA PSS optionally takes a tuple of *(hash algorithm, mask generating function and salt length)* as the mechanism parameter:

```
signature = private.sign(data,
                         mechanism=Mechanism.RSA_PKCS_PSS,
                         mechanism_param=(Mechanism.SHA_1,
                                          MGF.SHA1,
                                          20))
```

DSA

The default signing and verification mechanism for RSA is *DSA_SHA512*.

Other mechanisms are available:

Mechanism	Notes
DSA	No hashing. 20, 28, 32, 48 or 64 bits.
DSA_SHA*	DSA with SHAx message digesting.

```
# Given a private key `private`
signature = private.sign(data)

# Given a public key `public`
assert public.verify(data, signature)
```

The parameters *r* and *s* are concatenated together as a single byte string (each value is 20 bytes long for a total of 40 bytes). To convert to the ASN.1 encoding (e.g. as used by X.509) use `pkcs11.util.dsa.encode_dsa_signature()`. To convert from the ASN.1 encoding into PKCS #11 encoding use `pkcs11.util.ec.decode_dsa_signature()`.

ECDSA

The default signing and verification mechanism for ECDSA is *ECDSA_SHA512*.

Other mechanisms are available:

Mechanism	Notes
ECDSA	No hashing. Input truncated to 1024 bits.
ECDSA_SHA*	ECDSA with SHAx message digesting.

```
# Given a private key `private`
signature = private.sign(data)

# Given a public key `public`
assert public.verify(data, signature)
```

The parameters *r* and *s* are concatenated together as a single byte string (both values are the same length). To convert to the ASN.1 encoding (e.g. as used by X.509) use `pkcs11.util.ec.encode_ecdsa_signature()`. To convert from the ASN.1 encoding into PKCS #11 encoding use `pkcs11.util.ec.decode_ecdsa_signature()`.

5.1.8 Wrapping/Unwrapping

The expectation when using HSMs is that secret and private keys never leave the secure boundary of the HSM. However, there is a use case for transmitting secret and private keys over insecure mediums. We can do this using key wrapping.

Key wrapping is similar to encryption and decryption except instead of turning plaintext into crypttext it turns key objects into crypttext and vice versa.

Keys must be marked as *EXTRACTABLE* to remove them from the HSM, even wrapped.

Key wrapping mechanisms usually mirror encryption mechanisms.

AES

Default key wrapping mode is *AES_ECB*. ECB is considered safe for key wrapping due to the lack of repeating blocks. Other mechanisms, such as the new *AES_KEY_WRAP* (if available), are also possible..

The key we're wrapping can be any sensitive key, either a secret key or a private key. In this example we're extracting an AES secret key:

```
# Given two secret keys, `key1` and `key2`, we can extract an encrypted
# version of `key2`
crypttext = key1.wrap_key(key2)
```

Wrapping doesn't store any parameters about the keys. We must supply those to import the key.

```
key = key1.unwrap_key(ObjectClass.SECRET_KEY, KeyType.AES, crypttext)
```

DES2/3

Default key wrapping mode is *DES3_ECB*. ECB is considered safe for key wrapping due to the lack of repeating blocks. Other mechanisms are available.

Operation is the same as for *AES*.

RSA

The key we're wrapping can be any sensitive key, either a secret key or a private key. In this example we're extracting an AES secret key:

```
# Given a public key, `public`, and a secret key `key`, we can extract an
encrypted version of `key`
crypttext = public.wrap_key(key)
```

Wrapping doesn't store any parameters about the keys. We must supply those to import the key.

```
# Given a private key, `private`, matching `public` above we can decrypt
# and import `key`.
key = private.unwrap_key(ObjectClass.SECRET_KEY, KeyType.AES, crypttext)
```

5.1.9 Deriving Shared Keys

Warning: Key derivation mechanisms do not verify the authenticity of the other party. Your application should include a mechanism to verify the other user's public key is really from that user to avoid man-in-the-middle attacks.

Where possible use an existing protocol.

Diffie-Hellman

DH lets us derive a shared key using shared domain parameters, our private key and the other party's public key, which is passed as a mechanism parameter.

The default DH derivation mechanism is *DH_PKCS_DERIVE*, which uses the algorithm described in PKCS #3.

Note: Other DH derivation mechanisms including X9.42 derivation are not currently supported.

```
# Given our DH private key `private` and the other party's public key
# `other_public`
key = private.derive_key(
    KeyType.AES, 128,
    mechanism_param=other_public)
```

If the other user's public key was encoded using RFC 3279, we can decode this with `pkcs11.util.dh.decode_dh_public_key()`:

```
from pkcs11.util.dh import decode_dh_public_key

key = private.derive_key(
    KeyType.AES, 128,
    mechanism_param=decode_dh_public_key(encoded_public_key))
```

And we can encode our public key for them using `pkcs11.util.dh.encode_dh_public_key()`:

```
from pkcs11.util.dh import encode_dh_public_key

# Given our DH public key `public`
encoded_public_key = encode_dh_public_key(public)
```

The shared derived key can now be used for any appropriate mechanism.

If you want to extract the shared key from the HSM, you can mark the key as *EXTRACTABLE*:

```
key = private.derive_key(
    KeyType.AES, 128,
    mechanism_param=other_public,
    template={
        Attribute.SENSITIVE: False,
        Attribute.EXTRACTABLE: True,
    })
# This is our shared secret key
print(key[Attribute.VALUE])
```

EC Diffie-Hellman

ECDH is supported using the *ECDH1_DERIVE* mechanism, similar to plain DH, except that the mechanism parameter is a tuple consisting of 3 parameters:

- a key derivation function (KDF);
- a shared value; and
- the other user's public key.

The supported KDFs vary from device to device, check your HSM documentation. For `pkcs11.mechanisms.KDF.NULL` (the most widely supported KDF), the shared value must be *None*.

Note: Other ECDH derivation mechanisms including co-factor derivation and MQV derivation are not currently supported.

```
from pkcs11 import KeyType, KDF

# Given our DH private key `private` and the other party's public key
# `other_public`
key = private.derive_key(
    KeyType.AES, 128,
    mechanism_param=(KDF.NULL, None, other_public))
```

The value of the other user's public key should usually be a raw byte string however some implementations require a DER-encoded byte string (i.e. the same format as *EC_POINT*)[3]. Use the *encode_ec_point* parameter to `pkcs11.util.ec.decode_ec_public_key()`.

Implementation	Other user's *EC_POINT* encoding
SoftHSM v2	DER-encoded
Nitrokey HSM	Raw
Thales nCipher	?

If you want to extract the shared key from the HSM, you can mark the key as *EXTRACTABLE*:

```
key = private.derive_key(
    KeyType.AES, 128,
    mechanism_param=(KDF.NULL, None, other_public),
    template={
        Attribute.SENSITIVE: False,
        Attribute.EXTRACTABLE: True,
    })
# This is our shared secret key
print(key[Attribute.VALUE])
```

[3] The incompatibility comes from this being unspecified in earlier versions of PKCS #11, although why they made it a different format to *EC_POINT* is unclear.

5.1.10 Digesting and Hashing

PKCS #11 exposes the ability to hash or digest data via a number of mechanisms. For performance reasons, this is rarely done in the HSM, and is usually done in your process. The only advantage of using this function over `hashlib` is the ability to digest *pkcs11.Key* objects.

To digest a message (e.g. with SHA-256):

```
from pkcs11 import Mechanism

digest = session.digest(data, mechanism=Mechanism.SHA_256)
```

You can also pass an iterable of data:

```
with open(file_in, 'rb') as input:
    # A generator yielding chunks of the file
    chunks = iter(lambda: input.read(buffer_size), '')
    digest = session.digest(chunks, mechanism=Mechanism.SHA_512)
```

Or a key (if supported):

```
digest = session.digest(public_key, mechanism=Mechanism.SHA_1)
```

Or even a combination of keys and data:

```
digest = session.digest((b'HEADER', key), mechanism=Mechanism.SHA_1)
```

5.1.11 Certificates

Certificates can be stored in the HSM as objects. PKCS#11 is limited in its handling of certificates, and does not provide features like parsing of X.509 etc. These should be handled in an external library (e.g. *asn1crypto*. PKCS#11 will not set attributes on the certificate based on the *VALUE* and these must be specified when creating the object.

X.509

The following attributes are defined:

VALUE The complete X.509 certificate (BER-encoded)

SUBJECT The certificate subject (DER-encoded X.509 distinguished name)

ISSUER The certificate issuer (DER-encoded X.509 distinguished name)

SERIAL The certificate serial (DER-encoded integer)

Additionally an extended set of attributes can be stored if your HSM supports it:

START_DATE The certificate start date (notBefore)

END_DATE The certificate end date (notAfter)

HASH_OF_SUBJECT_PUBLIC_KEY The identifier of the subject's public key (bytes)

HASH_OF_ISSUER_PUBLIC_KEY The identifier of the issuer's public key (bytes)

Importing Certificates

`pkcs11.util.x509.decode_x509_certificate()` can be used to decode X.509 certificates for storage in the HSM:

```
from pkcs11.util.x509 import decode_x509_certificate

cert = self.session.create_object(decode_x509_certificate(b'DER encoded X.509 cert...
↪'))
```

Exporting Certificates

The full certificate is stored as *VALUE*. Any X.509 capable library can use this data, e.g. *asn1crypto* or *PyOpenSSL*.

OpenSSL:

```
import OpenSSL
from pkcs11 import Attribute, ObjectClass

for cert in session.get_objects({
    Attribute.CLASS: ObjectClass.CERTIFICATE,
}):
    # Convert from DER-encoded value to OpenSSL object
    cert = OpenSSL.crypto.load_certificate(
        OpenSSL.crypto.FILETYPE_ASN1,
        cert[Attribute.VALUE],
    )

    # Retrieve values from the certificate
    subject = cert.get_subject()

    # Convert to PEM format
    cert = OpenSSL.crypto.dump_certificate(
        OpenSSL.crypto.FILETYPE_PEM,
        cert
    )
```

asn1crypto:

```
from asn1crypto import pem, x509

der_bytes = cert[Attribute.VALUE]

# Load a certificate object from the DER-encoded value
cert = x509.Certificate.load(der_bytes)

# Write out a PEM encoded value
pem_bytes = pem.armor('CERTIFICATE', der_bytes)
```

5.2 API Reference

Section Contents

5.2.1 Classes

`pkcs11` defines a high-level, "Pythonic" interface to PKCS#11.

class `pkcs11.`**`lib`**(*so*)

Initialises the PKCS#11 library.

Only one PKCS#11 library can be initialised.

Parameters **so** (`str`) – Path to the PKCS#11 library to initialise.

`get_slots`(*token_present=False*)

Returns a list of PKCS#11 device slots known to this library.

Parameters **token_present** – If true, will limit the results to slots with a token present.

Return type list(*Slot*)

`get_tokens`(*token_label=None*, *token_serial=None*, *token_flags=None*, *slot_flags=None*, *mechanisms=None*)

Generator yielding PKCS#11 tokens matching the provided parameters.

See also `get_token()`.

Parameters

- **token_label** (`str`) – Optional token label.
- **token_serial** (`bytes`) – Optional token serial.
- **token_flags** (`TokenFlag`) – Optional bitwise token flags.
- **slot_flags** (`SlotFlag`) – Optional bitwise slot flags.

- **mechanisms** (*iter (Mechanism)*) – Optional required mechanisms.

Return type iter(*Token*)

get_token (*token_label=None, token_serial=None, token_flags=None, slot_flags=None, mechanisms=None*)
Returns a single token or raises either *pkcs11.exceptions.NoSuchToken* or *pkcs11.exceptions.MultipleTokensReturned*.

See also *get_tokens()*.

Parameters

- **token_label** (*str*) – Optional token label.
- **token_serial** (*bytes*) – Optional token serial.
- **token_flags** (TokenFlag) – Optional bitwise token flags.
- **slot_flags** (SlotFlag) – Optional bitwise slot flags.
- **mechanisms** (*iter (Mechanism)*) – Optional required mechanisms.

Return type *Token*

wait_for_slot_event (*blocking=True*)
Waits for a slot event such as a token insertion or removal to occur.

Returns the Slot on which the event was detected. If not blocking and no slot events were detected *pkcs11.exceptions.NoEvent* is raised.

Parameters blocking – If true, the thread will block.

Return type *Slot*

reinitialize ()
Reinitializes the loaded PKCS#11 library.

Return type None

cryptoki_version
PKCS#11 Cryptoki standard version (tuple).

manufacturer_id
Library vendor's name (str).

library_description
Description of the vendor's library (str).

library_version
Vendor's library version (tuple).

class pkcs11.**Slot**
A PKCS#11 device slot.

This object represents a physical or software slot exposed by PKCS#11. A slot has hardware capabilities, e.g. supported mechanisms and may has a physical or software *Token* installed.

slot_id = None
Slot identifier (opaque).

slot_description = None
Slot name (str).

manufacturer_id = None
Slot/device manufacturer's name (str).

hardware_version = None
Hardware version (`tuple`).

firmware_version = None
Firmware version (`tuple`).

flags = None
Capabilities of this slot (`SlotFlag`).

get_token()
Returns the token loaded into this slot.

Return type *Token*

get_mechanisms()
Returns the mechanisms supported by this device.

Return type set(*Mechanism*)

get_mechanism_info(*mechanism*)
Returns information about the mechanism.

Parameters **mechanism** (`Mechanism`) – mechanism to learn about

Return type *MechanismInfo*

class pkcs11.**Token**
A PKCS#11 token.

A token can be physically installed in a *Slot*, or a software token, depending on your PKCS#11 library.

slot = None
The *Slot* this token is installed in.

label = None
Label of this token (`str`).

serial = None
Serial number of this token (`bytes`).

manufacturer_id = None
Manufacturer ID.

model = None
Model name.

hardware_version = None
Hardware version (`tuple`).

firmware_version = None
Firmware version (`tuple`).

flags = None
Capabilities of this token (`pkcs11.flags.TokenFlag`).

open(*rw=False*, *user_pin=None*, *so_pin=None*)
Open a session on the token and optionally log in as a user or security officer (pass one of *user_pin* or *so_pin*). Pass PROTECTED_AUTH to indicate the pin should be supplied via an external mechanism (e.g. pin pad).

Can be used as a context manager or close with *Session.close()*.

```
with token.open() as session:

    print(session)
```

Parameters

- **rw** – True to create a read/write session.
- **user_pin** (*bytes*) – Authenticate to this session as a user.
- **so_pin** (*bytes*) – Authenticate to this session as a security officer.

Return type *Session*

class pkcs11.**Session**

A PKCS#11 `Token` session.

A session is required to do nearly all operations on a token including encryption/signing/keygen etc.

Create a session using `Token.open()`. Sessions can be used as a context manager or closed with `close()`.

token = None

`Token` this session is on.

rw = None

True if this is a read/write session.

user_type = None

User type for this session (`pkcs11.constants.UserType`).

close()

Close the session.

get_key(*object_class=None*, *key_type=None*, *label=None*, *id=None*)

Search for a key with any of *key_type*, *label* and/or *id*.

Returns a single key or throws `pkcs11.exceptions.NoSuchKey` or `pkcs11.exceptions.MultipleObjectsReturned`.

This is a simplified version of `get_objects()`, which allows searching for any object.

Parameters

- **object_class** (ObjectClass) – Optional object class.
- **key_type** (KeyType) – Optional key type.
- **label** (*str*) – Optional key label.
- **id** (*bytes*) – Optional key id.

Return type *Key*

get_objects(*attrs=None*)

Search for objects matching *attrs*. Returns a generator.

```
for obj in session.get_objects({
    Attribute.CLASS: ObjectClass.SECRET_KEY,
    Attribute.LABEL: 'MY LABEL',
}):

    print(obj)
```

This is the more generic version of `get_key()`.

Parameters **attrs** (*dict* (Attribute, *)) – Attributes to search for.

Return type iter(*Object*)

create_object (*attrs*)

Create a new object on the *Token*. This is a low-level interface to create any type of object and can be used for importing data onto the Token.

```
key = session.create_object({
    pkcs11.Attribute.CLASS: pkcs11.ObjectClass.SECRET_KEY,
    pkcs11.Attribute.KEY_TYPE: pkcs11.KeyType.AES,
    pkcs11.Attribute.VALUE: b'SUPER SECRET KEY',
})
```

For generating keys see *generate_key()* or *generate_keypair()*. For importing keys see *Importing/Exporting Keys*.

Requires a read/write session, unless the object is not to be stored. To permanently store the object in the HSM add **pkcs.Attribute.TOKEN: True**, see `pkcs11.Attribute()` for more available object attributes.

Parameters **attrs** (*dict* (Attribute, *)) – attributes of the object to create

Return type *Object*

create_domain_parameters (*key_type*, *attrs*, *local=False*, *store=False*)

Create a domain parameters object from known parameters.

Domain parameters are used for key generation of key types such as DH, DSA and EC.

You can also generate new parameters using *generate_domain_parameters()*.

The *local* parameter creates a Python object that is not created on the HSM (its object handle will be unset). This is useful if you only need the domain parameters to create another object, and do not need a real PKCS #11 object in the session.

> **Warning:** Domain parameters have no id or labels. Storing them is possible but be aware they may be difficult to retrieve.

Parameters

- **key_type** (KeyType) – Key type these parameters are for
- **attrs** (*dict* (Attribute, *)) – Domain parameters (specific tp *key_type*)
- **local** – if True, do not transfer parameters to the HSM.
- **store** – if True, store these parameters permanently in the HSM.

Return type *DomainParameters*

generate_domain_parameters (*key_type*, *param_length*, *store=False*, *mechanism=None*, *mechanism_param=None*, *template=None*)

Generate domain parameters.

See *create_domain_parameters()* for creating domain parameter objects from known parameters.

See *generate_key()* for documentation on mechanisms and templates.

> **Warning:** Domain parameters have no id or labels. Storing them is possible but be aware they may be difficult to retrieve.

Parameters

- **key_type** (`KeyType`) – Key type these parameters are for
- **params_length** (*int*) – Size of the parameters (e.g. prime length) in bits.
- **store** – Store these parameters in the HSM
- **mechanism** (`Mechanism`) – Optional generation mechanism (or default)
- **mechanism_param** (*bytes*) – Optional mechanism parameter.
- **template** (*dict (*`Attribute`*, *)*) – Optional additional attributes.

Return type *DomainParameters*

generate_key (*key_type, key_length=None, id=None, label=None, store=False, capabilities=None, mechanism=None, mechanism_param=None, template=None*)

Generate a single key (e.g. AES, DES).

Keys should set at least *id* or *label*.

An appropriate *mechanism* will be chosen for *key_type* (see `DEFAULT_GENERATE_MECHANISMS`) or this can be overridden. Similarly for the *capabilities* (see `DEFAULT_KEY_CAPABILITIES`).

The *template* will extend the default template used to make the key.

Possible mechanisms and template attributes are defined by PKCS #11. Invalid mechanisms or attributes should raise `pkcs11.exceptions.MechanismInvalid` and `pkcs11.exceptions.AttributeTypeInvalid` respectively.

Parameters

- **key_type** (`KeyType`) – Key type (e.g. KeyType.AES)
- **key_length** (*int*) – Key length in bits (e.g. 256).
- **id** (*bytes*) – Key identifier.
- **label** (*str*) – Key label.
- **store** – Store key on token (requires R/W session).
- **capabilities** (`MechanismFlag`) – Key capabilities (or default).
- **mechanism** (`Mechanism`) – Generation mechanism (or default).
- **mechanism_param** (*bytes*) – Optional vector to the mechanism.
- **template** (*dict (*`Attribute`*, *)*) – Additional attributes.

Return type *SecretKey*

generate_keypair (*key_type, key_length=None, **kwargs*)

Generate a asymmetric keypair (e.g. RSA).

See `generate_key()` for more information.

Parameters

- **key_type** (`KeyType`) – Key type (e.g. KeyType.DSA)
- **key_length** (*int*) – Key length in bits (e.g. 256).

- **id** (*bytes*) – Key identifier.
- **label** (*str*) – Key label.
- **store** (*bool*) – Store key on token (requires R/W session).
- **capabilities** (MechanismFlag) – Key capabilities (or default).
- **mechanism** (Mechanism) – Generation mechanism (or default).
- **mechanism_param** (*bytes*) – Optional vector to the mechanism.
- **private_template** (*dict (Attribute, *)*) – Additional attributes for private key.
- **public_template** (*dict (Attribute, *)*) – Additional attributes for public key.

Return type (*PublicKey*, *PrivateKey*)

seed_random (*seed*)

Mix additional seed material into the RNG (if supported).

Parameters **seed** (*bytes*) – Bytes of random to seed.

generate_random (*nbits*)

Generate *length* bits of random or pseudo-random data (if supported).

Parameters **nbits** (*int*) – Number of bits to generate.

Return type bytes

digest (*data*, ***kwargs*)

Digest *data* using *mechanism*.

data can be a single value or an iterator.

Key objects can also be digested, optionally interspersed with `bytes`.

Parameters

- **data** (*str, bytes, Key or iter(bytes, Key)*) – Data to digest
- **mechanism** (Mechanism) – digest mechanism
- **mechanism_param** (*bytes*) – optional mechanism parameter

Return type bytes

Token Objects

The following classes relate to *Object* objects on the *Token*.

class pkcs11.**Object**

A PKCS#11 object residing on a *Token*.

Objects implement `__getitem__()` and `__setitem__()` to retrieve *pkcs11.constants.Attribute* values on the object. Valid attributes for an object are given in PKCS #11. Invalid attributes should raise *pkcs11.exceptions.AttributeTypeInvalid*.

object_class = None

pkcs11.constants.ObjectClass of this Object.

session = None

Session this object is valid for.

copy (*attrs*)
Make a copy of the object with new attributes *attrs*.

Requires a read/write session, unless the object is not to be stored.

```
new = key.copy({
    Attribute.LABEL: 'MY NEW KEY',
})
```

Certain objects may not be copied. Calling *copy()* on such objects will result in an exception.

Parameters **attrs** (*dict* (Attribute, *)) – attributes for the new *Object*

Return type *Object*

destroy ()
Destroy the object.

Requires a read/write session, unless the object is not stored.

Certain objects may not be destroyed. Calling *destroy()* on such objects will result in an exception.

The *Object* is no longer valid.

class pkcs11.**Key** (*Object*)
Base class for all key *Object* types.

id
Key id (bytes).

label
Key label (str).

key_type
Key type (*pkcs11.mechanisms.KeyType*).

class pkcs11.**SecretKey** (*Key*)
A PKCS#11 *pkcs11.constants.ObjectClass.SECRET_KEY* object (symmetric encryption key).

key_length
Key length in bits.

class pkcs11.**PublicKey** (*Key*)
A PKCS#11 *pkcs11.constants.ObjectClass.PUBLIC_KEY* object (asymmetric public key).

RSA private keys can be imported and exported from PKCS#1 DER-encoding using *pkcs11.util.rsa.decode_rsa_public_key()* and *pkcs11.util.rsa.encode_rsa_public_key()* respectively.

key_length
Key length in bits.

class pkcs11.**PrivateKey** (*Key*)
A PKCS#11 *pkcs11.constants.ObjectClass.PRIVATE_KEY* object (asymmetric private key).

RSA private keys can be imported from PKCS#1 DER-encoding using *pkcs11.util.rsa.decode_rsa_private_key()*.

> **Warning:** Private keys imported directly, rather than unwrapped from a trusted private key should be considered insecure.

key_length
Key length in bits.

class pkcs11.**DomainParameters**(*Object*)
PKCS#11 Domain Parameters.

Used to store domain parameters as part of the key generation step, e.g. in DSA and Diffie-Hellman.

key_type
Key type (*pkcs11.mechanisms.KeyType*) these parameters can be used to generate.

generate_keypair(*id=None*, *label=None*, *store=False*, *capabilities=None*, *mechanism=None*, *mechanism_param=None*, *public_template=None*, *private_template=None*)
Generate a key pair from these domain parameters (e.g. for Diffie-Hellman.

See *Session.generate_key()* for more information.

Parameters

- **id** (*bytes*) – Key identifier.
- **label** (*str*) – Key label.
- **store** – Store key on token (requires R/W session).
- **capabilities** (MechanismFlag) – Key capabilities (or default).
- **mechanism** (Mechanism) – Generation mechanism (or default).
- **mechanism_param** (*bytes*) – Optional vector to the mechanism.
- **template** (*dict* (Attribute, *)) – Additional attributes.

Return type (*PublicKey*, *PrivateKey*)

class pkcs11.**Certificate**(*Object*)
A PKCS#11 *pkcs11.constants.ObjectClass.CERTIFICATE* object.

PKCS#11 is limited in its handling of certificates, and does not provide features like parsing of X.509 etc. These should be handled in an external library. PKCS#11 will not set attributes on the certificate based on the *VALUE*.

pkcs11.util.x509.decode_x509_certificate() will extract attributes from a certificate to create the object.

certificate_type
The type of certificate.

Return type *CertificateType*

Object Capabilities

Capability mixins for *Object* objects.

class pkcs11.**EncryptMixin**
This *Object* supports the encrypt capability.

encrypt(*data*, *buffer_size=8192*, ***kwargs*)
Encrypt some *data*.

Data can be either str or bytes, in which case it will return bytes; or an iterable of bytes in which case it will return a generator yielding bytes (be aware, more chunks will be output than input).

If you do not specify *mechanism* then the default from DEFAULT_ENCRYPT_MECHANISMS will be used. If an iterable is passed and the mechanism chosen does not support handling data in chunks, an exception will be raised.

Some mechanisms (including the default CBC mechanisms) require additional parameters, e.g. an initialisation vector[1], to the mechanism. Pass this as *mechanism_param*. Documentation of these parameters is given specified in PKCS #11.

When passing an iterable for data *buffer_size* must be sufficient to store the working buffer. An integer number of blocks and greater than or equal to the largest input chunk is recommended.

The returned generator obtains a lock on the `Session` to prevent other threads from starting a simultaneous operation. The lock is released when you consume/destroy the generator. See *Concurrency*.

> **Warning:** It's not currently possible to cancel an encryption operation by deleting the generator. You must consume the generator to complete the operation.

An example of streaming a file is as follows:

```
def encrypt_file(file_in, file_out, buffer_size=8192):

    with \
            open(file_in, 'rb') as input_, \
            open(file_out, 'wb') as output:

        chunks = iter(lambda: input_.read(buffer_size), '')

        for chunk in key.encrypt(chunks,
                                 mechanism_param=iv,
                                 buffer_size=buffer_size):
            output.write(chunk)
```

Parameters

- **data** (*str, bytes or iter(bytes)*) – data to encrypt
- **mechanism** (Mechanism) – optional encryption mechanism (or None for default)
- **mechanism_param** (*bytes*) – optional mechanism parameter (e.g. initialisation vector).
- **buffer_size** (*int*) – size of the working buffer (for generators)

Return type bytes or iter(bytes)

class pkcs11.**DecryptMixin**

This `Object` supports the decrypt capability.

decrypt (*data*, *buffer_size=8192*, ***kwargs*)

Decrypt some *data*.

See `EncryptMixin.encrypt()` for more information.

Parameters

- **data** (*bytes or iter(bytes)*) – data to decrypt
- **mechanism** (Mechanism) – optional encryption mechanism (or None for default).
- **mechanism_param** (*bytes*) – optional mechanism parameter (e.g. initialisation vector).

[1] The initialisation vector should contain quality random, e.g. from `Session.generate_random()`. This method will not return the value of the initialisation vector as part of the encryption. You must store that yourself.

- **buffer_size** (*int*) – size of the working buffer (for generators).

Return type bytes or iter(bytes)

class pkcs11.**SignMixin**

This *Object* supports the sign capability.

sign (*data*, ***kwargs*)

Sign some *data*.

See *EncryptMixin.encrypt()* for more information.

For DSA and ECDSA keys, PKCS #11 outputs the two parameters (r & s) as two concatenated *biginteger* of the same length. To convert these into other formats, such as the format used by OpenSSL, use *pkcs11.util.dsa.encode_dsa_signature()* or *pkcs11.util.ec.encode_ecdsa_signature()*.

Parameters

- **data** (*str, bytes or iter(bytes)*) – data to sign
- **mechanism** (Mechanism) – optional signing mechanism
- **mechanism_param** (*bytes*) – optional mechanism parameter

Return type bytes

class pkcs11.**VerifyMixin**

This *Object* supports the verify capability.

verify (*data*, *signature*, ***kwargs*)

Verify some *data*.

See *EncryptMixin.encrypt()* for more information.

Returns True if *signature* is valid for *data*.

For DSA and ECDSA keys, PKCS #11 expects the two parameters (r & s) as two concatenated *biginteger* of the same length. To convert these from other formats, such as the format used by OpenSSL, use *pkcs11.util.dsa.decode_dsa_signature()* or *pkcs11.util.ec.decode_ecdsa_signature()*.

Parameters

- **data** (*str, bytes or iter(bytes)*) – data to sign
- **signature** (*bytes*) – signature
- **mechanism** (Mechanism) – optional signing mechanism
- **mechanism_param** (*bytes*) – optional mechanism parameter

Return type bool

class pkcs11.**WrapMixin**

This *Object* supports the wrap capability.

wrap_key (*key*, *mechanism=None*, *mechanism_param=None*)

Use this key to wrap (i.e. encrypt) *key* for export. Returns an encrypted version of *key*.

key must have `Attribute.EXTRACTABLE` = True.

Parameters

- **key** (Key) – key to export
- **mechanism** (Mechanism) – wrapping mechanism (or None for default).

- **mechanism_param** (*bytes*) – mechanism parameter (if required)

Return type bytes

class pkcs11.**UnwrapMixin**

This *Object* supports the unwrap capability.

unwrap_key (*object_class, key_type, key_data, id=None, label=None, mechanism=None, mechanism_param=None, store=False, capabilities=None, template=None*)

Use this key to unwrap (i.e. decrypt) and import *key_data*.

See *Session.generate_key* for more information.

Parameters

- **object_class** (ObjectClass) – Object class to import as
- **key_type** (KeyType) – Key type (e.g. KeyType.AES)
- **key_data** (*bytes*) – Encrypted key to unwrap
- **id** (*bytes*) – Key identifier.
- **label** (*str*) – Key label.
- **store** – Store key on token (requires R/W session).
- **capabilities** (MechanismFlag) – Key capabilities (or default).
- **mechanism** (Mechanism) – Generation mechanism (or default).
- **mechanism_param** (*bytes*) – Optional vector to the mechanism.
- **template** (*dict (*Attribute*, *)*) – Additional attributes.

Return type *Key*

class pkcs11.**DeriveMixin**

This *Object* supports the derive capability.

derive_key (*key_type, key_length, id=None, label=None, store=False, capabilities=None, mechanism=None, mechanism_param=None, template=None*)

Derive a new key from this key. Used to create session keys from a PKCS key exchange.

Typically the mechanism, e.g. Diffie-Hellman, requires you to specify the other party's piece of shared information as the *mechanism_param*. Some mechanisms require a tuple of data (see *pkcs11.mechanisms.Mechanism*).

See *Session.generate_key* for more documentation on key generation.

Diffie-Hellman example:

```
# Diffie-Hellman domain parameters
# e.g. from RFC 3526, RFC 5114 or `openssl dhparam`
prime = [0xFF, ...]
base = [0x02]

parameters = session.create_domain_parameters(KeyType.DH, {
    Attribute.PRIME: prime,
    Attribute.BASE: base,
}, local=True)

# Alice generates a DH key pair from the public
# Diffie-Hellman parameters
public, private = parameters.generate_keypair()
```

(continues on next page)

(continued from previous page)

```
alices_value = public[Attribute.VALUE]

# Bob generates a DH key pair from the same parameters.

# Alice exchanges public values with Bob...
# She sends `alices_value` and receives `bobs_value`.
# (Assuming Alice is doing AES CBC, she also needs to send an IV)

# Alice generates a session key with Bob's public value
# Bob will generate the same session key using Alice's value.
session_key = private.derive_key(
    KeyType.AES, 128,
    mechanism_param=bobs_value)
```

Elliptic-Curve Diffie-Hellman example:

```
# DER encoded EC params, e.g. from OpenSSL
# openssl ecparam -outform der -name prime192v1 | base64
#
# Check what EC parameters the module supports with
# slot.get_module_info()
parameters = session.create_domain_parameters(KeyType.EC, {
    Attribute.EC_PARAMS: b'...',
}, local=True)

# Alice generates a EC key pair, and gets her public value
public, private = parameters.generate_keypair()
alices_value = public[Attribute.EC_POINT]

# Bob generates a DH key pair from the same parameters.

# Alice exchanges public values with Bob...
# She sends `alices_value` and receives `bobs_value`.

# Alice generates a session key with Bob's public value
# Bob will generate the same session key using Alice's value.
session_key = private.derive_key(
    KeyType.AES, 128,
    mechanism_param=(KDF.NULL, None, bobs_value))
```

Parameters

- **key_type** (KeyType) – Key type (e.g. KeyType.AES)
- **key_length** (*int*) – Key length in bits (e.g. 256).
- **id** (*bytes*) – Key identifier.
- **label** (*str*) – Key label.
- **store** – Store key on token (requires R/W session).
- **capabilities** (MechanismFlag) – Key capabilities (or default).
- **mechanism** (Mechanism) – Generation mechanism (or default).
- **mechanism_param** (*bytes*) – Optional vector to the mechanism.
- **template** (*dict (*Attribute*, *)*) – Additional attributes.

Return type *SecretKey*

5.2.2 Constants

PKCS#11 constants.

See the Python `enum` documentation for more information on how to use these classes.

`pkcs11.constants.DEFAULT = <object object>`

Sentinel value used in templates.

Not all pkcs11 attribute sets are accepted by HSMs. Use this value to remove the attribute from the template sent to the HSM or to use the HSM default value.

class `pkcs11.constants.UserType`

PKCS#11 user types.

NOBODY = 999

Not officially in the PKCS#11 spec. Used to represent a session that is not logged in.

SO = 0

Security officer.

USER = 1

class `pkcs11.constants.ObjectClass`

PKCS#11 `Object` class.

This is the type of object we have.

DATA = 0

CERTIFICATE = 1

See *pkcs11.Certificate*.

PUBLIC_KEY = 2

See *pkcs11.PublicKey*.

PRIVATE_KEY = 3

See *pkcs11.PrivateKey*.

SECRET_KEY = 4

See *pkcs11.SecretKey*.

HW_FEATURE = 5

DOMAIN_PARAMETERS = 6

See *pkcs11.DomainParameters*.

MECHANISM = 7

OTP_KEY = 8

class `pkcs11.constants.Attribute`

PKCS#11 object attributes.

Not all attributes are relevant to all objects. Relevant attributes for each object type are given in PKCS #11.

CLASS = 0

Object type (*ObjectClass*).

TOKEN = 1

If True object will be stored to token. Otherwise has session lifetime (`bool`).

PRIVATE = 2
True if user must be authenticated to access this object (`bool`).

LABEL = 3
Object label (`str`).

APPLICATION = 16

VALUE = 17
Object value. Usually represents a secret or private key. For certificates this is the complete certificate in the certificate's native format (e.g. BER-encoded X.509 or WTLS encoding).

May be *SENSITIVE* (`bytes`).

OBJECT_ID = 18

CERTIFICATE_TYPE = 128
Certificate type (*`CertificateType`*).

ISSUER = 129
Certificate issuer in certificate's native format (e.g. X.509 DER-encoding or WTLS encoding) (`bytes`).

SERIAL_NUMBER = 130
Certificate serial number in certificate's native format (e.g. X.509 DER-encoding) (`bytes`).

AC_ISSUER = 131
Attribute Certificate Issuer. Different from *ISSUER* because the encoding is different (`bytes`).

OWNER = 132
Attribute Certificate Owner. Different from *SUBJECT* because the encoding is different (`bytes`).

ATTR_TYPES = 133
BER-encoding of a sequence of object identifier values corresponding to the attribute types contained in the certificate. When present, this field offers an opportunity for applications to search for a particular attribute certificate without fetching and parsing the certificate itself.

TRUSTED = 134
This key can be used to wrap keys with *WRAP_WITH_TRUSTED* set; or this certificate can be trusted. (`bool`).

CERTIFICATE_CATEGORY = 135
Certificate category (`CertificateCategory`).

JAVA_MIDP_SECURITY_DOMAIN = 136

URL = 137
URL where the complete certificate can be obtained.

HASH_OF_SUBJECT_PUBLIC_KEY = 138
Hash of the certificate subject's public key.

HASH_OF_ISSUER_PUBLIC_KEY = 139
Hash of the certificate issuer's public key.

CHECK_VALUE = 144
VALUE checksum. Key Check Value (`bytes`).

KEY_TYPE = 256
Key type (`KeyType`).

SUBJECT = 257
Certificate subject in certificate's native format (e.g. X.509 DER-encoding or WTLS encoding) (`bytes`).

ID = 258
Key ID (bytes).

SENSITIVE = 259
Sensitive attributes cannot be retrieved from the HSM (e.g. *VALUE* or *PRIVATE_EXPONENT*) (`bool`).

ENCRYPT = 260
Key supports encryption (`bool`).

DECRYPT = 261
Key supports decryption (`bool`).

WRAP = 262
Key supports wrapping (`bool`).

UNWRAP = 263
Key supports unwrapping (`bool`).

SIGN = 264
Key supports signing (`bool`).

SIGN_RECOVER = 265

VERIFY = 266
Key supports signature verification (`bool`).

VERIFY_RECOVER = 267

DERIVE = 268
Key supports key derivation (`bool`).

START_DATE = 272
Start date for the object's validity (`datetime.date`).

END_DATE = 273
End date for the object's validity (`datetime.date`).

MODULUS = 288
RSA private key modulus (n) (*biginteger* as `bytes`).

MODULUS_BITS = 289
RSA private key modulus length. Use this for private key generation (`int`).

PUBLIC_EXPONENT = 290
RSA public exponent (e) (*biginteger* as `bytes`).

Default is b'' (65537).

PRIVATE_EXPONENT = 291
RSA private exponent (d) (*biginteger* as `bytes`).

PRIME_1 = 292
RSA private key prime #1 (p). May not be stored. (*biginteger* as `bytes`).

PRIME_2 = 293
RSA private key prime #2 (q). May not be stored. (*biginteger* as `bytes`).

EXPONENT_1 = 294
RSA private key exponent #1 (d mod p-1). May not be stored. (*biginteger* as `bytes`).

EXPONENT_2 = 295
RSA private key exponent #2 (d mod q-1). May not be stored. (*biginteger* as `bytes`).

COEFFICIENT = 296
RSA private key CRT coefficient (q^-1 mod p). May not be stored. (*biginteger* as `bytes`).

PRIME = 304
Prime number 'q' (used for DH). (*biginteger* as `bytes`).

SUBPRIME = 305
Subprime number 'q' (used for DH). (*biginteger* as `bytes`).

BASE = 306
Base number 'g' (used for DH). (*biginteger* as `bytes`).

PRIME_BITS = 307

SUBPRIME_BITS = 308

VALUE_BITS = 352

VALUE_LEN = 353
VALUE length in bytes. Use this for secret key generation (`int`).

EXTRACTABLE = 354
Key can be extracted wrapped.

LOCAL = 355
True if generated on the token, False if imported.

NEVER_EXTRACTABLE = 356
EXTRACTABLE has always been False.

ALWAYS_SENSITIVE = 357
SENSITIVE has always been True.

KEY_GEN_MECHANISM = 358
Key generation mechanism (*pkcs11.mechanisms.Mechanism*).

MODIFIABLE = 368
Object can be modified (`bool`).

COPYABLE = 369
Object can be copied (`bool`).

EC_PARAMS = 384
DER-encoded ANSI X9.62 Elliptic-Curve domain parameters (`bytes`).

These can packed using *pkcs11.util.ec.encode_named_curve_parameters*:

```
from pkcs11.util.ec import encode_named_curve_parameters

ecParams = encode_named_curve_parameters('secp256r1')
```

Or output by OpenSSL:

```
openssl ecparam -outform der -name <curve name> | base64
```

EC_POINT = 385
DER-encoded ANSI X9.62 Public key for `KeyType.EC` (`bytes`).

SECONDARY_AUTH = 512

AUTH_PIN_FLAGS = 513

ALWAYS_AUTHENTICATE = 514
User has to provide pin with each use (sign or decrypt) (`bool`).

WRAP_WITH_TRUSTED = 528
Key can only be wrapped with a *TRUSTED* key.

```
WRAP_TEMPLATE = 1073742353

UNWRAP_TEMPLATE = 1073742354

DERIVE_TEMPLATE = 1073742355

OTP_FORMAT = 544

OTP_LENGTH = 545

OTP_TIME_INTERVAL = 546

OTP_USER_FRIENDLY_MODE = 547

OTP_CHALLENGE_REQUIREMENT = 548

OTP_TIME_REQUIREMENT = 549

OTP_COUNTER_REQUIREMENT = 550

OTP_PIN_REQUIREMENT = 551

OTP_COUNTER = 558

OTP_TIME = 559

OTP_USER_IDENTIFIER = 554

OTP_SERVICE_IDENTIFIER = 555

OTP_SERVICE_LOGO = 556

OTP_SERVICE_LOGO_TYPE = 557

GOSTR3410_PARAMS = 592

GOSTR3411_PARAMS = 593

GOST28147_PARAMS = 594

HW_FEATURE_TYPE = 768

RESET_ON_INIT = 769

HAS_RESET = 770

PIXEL_X = 1024

PIXEL_Y = 1025

RESOLUTION = 1026

CHAR_ROWS = 1027

CHAR_COLUMNS = 1028

COLOR = 1029

BITS_PER_PIXEL = 1030

CHAR_SETS = 1152

ENCODING_METHODS = 1153

MIME_TYPES = 1154

MECHANISM_TYPE = 1280

REQUIRED_CMS_ATTRIBUTES = 1281

DEFAULT_CMS_ATTRIBUTES = 1282
```

SUPPORTED_CMS_ATTRIBUTES = 1283

ALLOWED_MECHANISMS = 1073743360

class pkcs11.constants.**CertificateType**

Certificate type of a *pkcs11.Certificate*.

X_509 = 0

X_509_ATTR_CERT = 1

WTLS = 2

class pkcs11.constants.**MechanismFlag**

Describes the capabilities of a *pkcs11.mechanisms.Mechanism* or *pkcs11.Object*.

Some objects and mechanisms are symmetric (i.e. can be used for encryption and decryption), some are asymmetric (e.g. public key cryptography).

HW = 1

Mechanism is performed in hardware.

ENCRYPT = 256

Can be used for encryption.

DECRYPT = 512

Can be used for decryption.

DIGEST = 1024

Can make a message digest (hash).

SIGN = 2048

Can calculate digital signature.

SIGN_RECOVER = 4096

VERIFY = 8192

Can verify digital signature.

VERIFY_RECOVER = 16384

GENERATE = 32768

Can generate key/object.

GENERATE_KEY_PAIR = 65536

Can generate key pair.

WRAP = 131072

Can wrap a key for export.

UNWRAP = 262144

Can unwrap a key for import.

DERIVE = 524288

Can derive a key from another key.

EC_F_P = 1048576

EC_F_2M = 2097152

EC_ECPARAMETERS = 4194304

EC_NAMEDCURVE = 8388608

EC_UNCOMPRESS = 16777216

EC_COMPRESS = 33554432

EXTENSION = 2147483648

class pkcs11.constants.**SlotFlag**

pkcs11.Slot flags.

TOKEN_PRESENT = 1

A token is present in the slot (N.B. some hardware known not to set this for soft-tokens.)

REMOVABLE_DEVICE = 2

Removable devices.

HW_SLOT = 4

Hardware slot.

class pkcs11.constants.**TokenFlag**

pkcs11.Token flags.

RNG = 1

Has random number generator.

WRITE_PROTECTED = 2

Token is write protected.

LOGIN_REQUIRED = 4

User must login.

USER_PIN_INITIALIZED = 8

Normal user's pin is set.

RESTORE_KEY_NOT_NEEDED = 32

If it is set, that means that *every* time the state of cryptographic operations of a session is successfully saved, all keys needed to continue those operations are stored in the state.

CLOCK_ON_TOKEN = 64

If it is set, that means that the token has some sort of clock. The time on that clock is returned in the token info structure.

PROTECTED_AUTHENTICATION_PATH = 256

If it is set, that means that there is some way for the user to login without sending a PIN through the Cryptoki library itself.

DUAL_CRYPTO_OPERATIONS = 512

If it is true, that means that a single session with the token can perform dual simultaneous cryptographic operations (digest and encrypt; decrypt and digest; sign and encrypt; and decrypt and sign).

TOKEN_INITIALIZED = 1024

If it is true, the token has been initialized using C_InitializeToken or an equivalent mechanism outside the scope of PKCS #11. Calling C_InitializeToken when this flag is set will cause the token to be reinitialized.

USER_PIN_COUNT_LOW = 65536

If it is true, an incorrect user login PIN has been entered at least once since the last successful authentication.

USER_PIN_FINAL_TRY = 131072

If it is true, supplying an incorrect user PIN will it to become locked.

USER_PIN_LOCKED = 262144

If it is true, the user PIN has been locked. User login to the token is not possible.

USER_PIN_TO_BE_CHANGED = 524288

If it is true, the user PIN value is the default value set by token initialization or manufacturing, or the PIN has been expired by the card.

SO_PIN_COUNT_LOW = 1048576
If it is true, an incorrect SO (security officer) login PIN has been entered at least once since the last successful authentication.

SO_PIN_FINAL_TRY = 2097152
If it is true, supplying an incorrect SO (security officer) PIN will it to become locked.

SO_PIN_LOCKED = 4194304
If it is true, the SO (security officer) PIN has been locked. SO login to the token is not possible.

SO_PIN_TO_BE_CHANGED = 8388608
If it is true, the SO PIN value is the default value set by token initialization or manufacturing, or the PIN has been expired by the card.

ERROR_STATE = 16777216

5.2.3 Key Types & Mechanisms

class pkcs11.mechanisms.**KeyType**
Key types known by PKCS#11.

Making use of a given key type requires the appropriate *Mechanism* to be available.

Key types beginning with an underscore are historic and are best avoided.

RSA = 0
See the RSA section of the PKCS #11 specification for valid *Mechanism* and *pkcs11.constants.Attribute* types.

DSA = 1
See the DSA section of the PKCS #11 specification for valid *Mechanism* and *pkcs11.constants.Attribute* types.

DH = 2
PKCS #3 Diffie-Hellman key. See the Diffie-Hellman section of the PKCS #11 specification for valid *Mechanism* and *pkcs11.constants.Attribute* types.

EC = 3
See the Elliptic Curve section of the PKCS #11 specification for valid *Mechanism* and *pkcs11.constants.Attribute* types.

X9_42_DH = 4
X9.42 Diffie-Hellman key.

GENERIC_SECRET = 16

DES2 = 20

> **Warning:** Considered insecure. Use AES where possible.

DES3 = 21

> **Warning:** Considered insecure. Use AES where possible.

AES = 31
See the AES section of PKCS#11 for valid *Mechanism* and *pkcs11.constants.Attribute* types.

BLOWFISH = 32

TWOFISH = 33

SECURID = 34

HOTP = 35

ACTI = 36

CAMELLIA = 37

ARIA = 38

SHA_1_HMAC = 40

> **Warning:** SHA-1 is no longer considered secure.

SHA256_HMAC = 43

SHA384_HMAC = 44

SHA512_HMAC = 45

SHA224_HMAC = 46

SEED = 47

GOSTR3410 = 48

GOSTR3411 = 49

GOST28147 = 50

EC_EDWARDS = 64

class pkcs11.mechanisms.**Mechanism**
Cryptographic mechanisms known by PKCS#11.

The list of supported cryptographic mechanisms for a *pkcs11.Slot* can be retrieved with *pkcs11.Slot.get_mechanisms()*.

Mechanisms beginning with an underscore are historic and best avoided. Descriptions of the current and historical mechanisms, including their valid *pkcs11.constants.Attribute* types and *mechanism_param* can be found in the PKCS#11 specification.

Additionally, while still in the *current* spec, a number of mechanisms including cryptographic hash functions and certain block modes are no longer considered secure, and should not be used for new applications, e.g. MD2, MD5, SHA1, ECB.

RSA_PKCS_KEY_PAIR_GEN = 0
RSA PKCS #1 v1.5 key generation.

> **Note:** Default for generating *KeyType.RSA* keys.

RSA_PKCS = 1
RSA PKCS #1 v1.5 general purpose mechanism.

> **Warning:** Consider using the more robust PKCS#1 OAEP.

RSA_PKCS_TPM_1_1 = 16385

> **Warning:** Consider using the more robust PKCS#1 OAEP.

RSA_PKCS_OAEP = 9

RSA PKCS #1 OAEP (v2.0+)

> **Note:** Default for encrypting/decrypting with `KeyType.RSA` keys.

Optionally takes a *mechanism_param* which is a tuple of:

- message digest algorithm used to calculate the digest of the encoding parameter (`Mechanism`), default is `Mechanism.SHA_1`;
- mask generation function to use on the encoded block (`MGF`), default is `MGF.SHA1`;
- data used as the input for the encoding parameter source (`bytes`), default is None.

RSA_PKCS_OAEP_TPM_1_1 = 16386

RSA_X_509 = 3

X.509 (raw) RSA.

No padding, supply your own.

RSA_9796 = 2

ISO/IEC 9796 RSA.

> **Warning:** DS1 and DS3 are considered broken. The PKCS #11 spec doesn't specify which scheme is used. Use *PSS* instead.

MD2_RSA_PKCS = 4

> **Warning:** Not considered secure.

MD5_RSA_PKCS = 5

> **Warning:** Not considered secure.

SHA1_RSA_PKCS = 6

> **Warning:** SHA-1 is no longer considered secure.

SHA224_RSA_PKCS = 70

SHA256_RSA_PKCS = 64

SHA384_RSA_PKCS = 65

SHA512_RSA_PKCS = 66

> **Note:** Default for signing/verification with `KeyType.RSA` keys.

RSA_PKCS_PSS = 13

RSA PSS without hashing.

PSS schemes optionally take a tuple of:

- message digest algorithm used to calculate the digest of the encoding parameter (`Mechanism`), default is `Mechanism.SHA_1`;
- mask generation function to use on the encoded block (`MGF`), default is `MGF.SHA1`; and
- salt length, default is 20

SHA1_RSA_PKCS_PSS = 14

> **Warning:** SHA-1 is no longer considered secure.

SHA224_RSA_PKCS_PSS = 71

SHA256_RSA_PKCS_PSS = 67

SHA384_RSA_PKCS_PSS = 68

SHA512_RSA_PKCS_PSS = 69

RSA_X9_31_KEY_PAIR_GEN = 10

RSA_X9_31 = 11

SHA1_RSA_X9_31 = 12

> **Warning:** SHA-1 is no longer considered secure.

DSA_KEY_PAIR_GEN = 16

> **Note:** Default mechanism for generating `KeyType.DSA` keypairs

Requires `pkcs11.DomainParameters`.

DSA = 17

DSA without hashing.

DSA_SHA1 = 18

> **Warning:** SHA-1 is no longer considered secure.

DSA_SHA224 = 19

DSA_SHA256 = 20

DSA_SHA384 = 21

DSA_SHA512 = 22

DSA with SHA512 hashing.

> **Note:** Default for signing/verification with *KeyType.DSA* keys.

DH_PKCS_KEY_PAIR_GEN = 32

> **Note:** Default mechanism for generating *KeyType.DH* key pairs.

This is the mechanism defined in PKCS #3.

Requires *pkcs11.DomainParameters* of *pkcs11.constants.Attribute.BASE* and *pkcs11.constants.Attribute.PRIME*.

DH_PKCS_DERIVE = 33

> **Note:** Default mechanism for deriving shared keys from *KeyType.DH* private keys.

This is the mechanism defined in PKCS #3.

Takes the other participant's public key *pkcs11.constants.Attribute.VALUE* as the *mechanism_param*.

X9_42_DH_KEY_PAIR_GEN = 48

X9_42_DH_DERIVE = 49

X9_42_DH_HYBRID_DERIVE = 50

X9_42_MQV_DERIVE = 51

DES2_KEY_GEN = 304

> **Note:** Default for generating DES2 keys.

> **Warning:** Considered insecure. Use AES where possible.

DES3_KEY_GEN = 305

> **Note:** Default for generating DES3 keys.

> **Warning:** Considered insecure. Use AES where possible.

DES3_ECB = 306

> **Note:** Default for key wrapping with DES2/3.

> **Warning:** Identical blocks will encipher to the same result. Considered insecure. Use AES where possible.

DES3_CBC = 307

DES3_MAC = 308

> **Note:** This is the default for signing/verification with *KeyType.DES2* and *KeyType.DES3*.

> **Warning:** Considered insecure. Use AES where possible.

DES3_MAC_GENERAL = 309

DES3_CBC_PAD = 310

> **Note:** Default for encryption/decryption with DES2/3.

> **Warning:** Considered insecure. Use AES where possible.

DES3_CMAC_GENERAL = 311

DES3_CMAC = 312

SHA_1 = 544

> **Warning:** SHA-1 is no longer considered secure.

SHA_1_HMAC = 545

> **Warning:** SHA-1 is no longer considered secure.

SHA_1_HMAC_GENERAL = 546

> **Warning:** SHA-1 is no longer considered secure.

SHA256 = 592

SHA256_HMAC = 593

SHA256_HMAC_GENERAL = 594

SHA224 = 597

SHA224_HMAC = 598

SHA224_HMAC_GENERAL = 599

SHA384 = 608

SHA384_HMAC = 609

SHA384_HMAC_GENERAL = 610

SHA512 = 624

SHA512_HMAC = 625

SHA512_HMAC_GENERAL = 626

SECURID_KEY_GEN = 640

SECURID = 642

HOTP_KEY_GEN = 656

HOTP = 657

ACTI = 672

ACTI_KEY_GEN = 673

GENERIC_SECRET_KEY_GEN = 848

CONCATENATE_BASE_AND_KEY = 864

CONCATENATE_BASE_AND_DATA = 866

CONCATENATE_DATA_AND_BASE = 867

XOR_BASE_AND_DATA = 868

EXTRACT_KEY_FROM_KEY = 869

SSL3_PRE_MASTER_KEY_GEN = 880

SSL3_MASTER_KEY_DERIVE = 881

SSL3_KEY_AND_MAC_DERIVE = 882

SSL3_MASTER_KEY_DERIVE_DH = 883

SSL3_MD5_MAC = 896

SSL3_SHA1_MAC = 897

TLS_PRE_MASTER_KEY_GEN = 884

TLS_MASTER_KEY_DERIVE = 885

TLS_KEY_AND_MAC_DERIVE = 886

TLS_MASTER_KEY_DERIVE_DH = 887

TLS_PRF = 888

SHA1_KEY_DERIVATION = 914

SHA256_KEY_DERIVATION = 915

SHA384_KEY_DERIVATION = 916

SHA512_KEY_DERIVATION = 917

SHA224_KEY_DERIVATION = 918

PKCS5_PBKD2 = 944

WTLS_PRE_MASTER_KEY_GEN = 976

WTLS_MASTER_KEY_DERIVE = 977

WTLS_MASTER_KEY_DERIVE_DH_ECC = 978

WTLS_PRF = 979

WTLS_SERVER_KEY_AND_MAC_DERIVE = 980

WTLS_CLIENT_KEY_AND_MAC_DERIVE = 981

CMS_SIG = 1280

KIP_DERIVE = 1296

KIP_WRAP = 1297

KIP_MAC = 1298

SEED_KEY_GEN = 1616

SEED_ECB = 1617

> **Warning:** Identical blocks will encipher to the same result.

SEED_CBC = 1618

SEED_MAC = 1619

SEED_MAC_GENERAL = 1620

SEED_CBC_PAD = 1621

SEED_ECB_ENCRYPT_DATA = 1622

SEED_CBC_ENCRYPT_DATA = 1623

EC_KEY_PAIR_GEN = 4160

> **Note:** Default mechanism for generating *KeyType.EC* key pairs

Requires *pkcs11.DomainParameters* of *pkcs11.constants.Attribute.EC_PARAMS*.

ECDSA = 4161
ECDSA with no hashing. Input truncated to 1024-bits.

ECDSA_SHA1 = 4162

> **Warning:** SHA-1 is no longer considered secure.

ECDSA_SHA224 = 4163

ECDSA_SHA256 = 4164

ECDSA_SHA384 = 4165

ECDSA_SHA512 = 4166
ECDSA with SHA512 hashing.

> **Note:** Default for signing/verification with *KeyType.EC* keys.

ECDH1_DERIVE = 4176

> **Note:** Default mechanism for deriving shared keys from *KeyType.EC* private keys.

Takes a tuple of:

- key derivation function (*pkcs11.mechanisms.KDF*);
- shared value (`bytes`); and
- other participant's *pkcs11.constants.Attribute.EC_POINT* (`bytes`)

as the *mechanism_param*.

ECDH1_COFACTOR_DERIVE = 4177

ECMQV_DERIVE = 4178

AES_KEY_GEN = 4224

> **Note:** Default for generating *KeyType.AES* keys.

AES_ECB = 4225

Note: Default wrapping mechanism for *KeyType.AES* keys.

Warning: Identical blocks will encipher to the same result.

AES_CBC = 4226

AES_CBC_PAD = 4229
CBC with PKCS#7 padding to pad files to a whole number of blocks.

Note: Default for encrypting/decrypting with *KeyType.AES* keys.

Requires a 128-bit initialisation vector passed as *mechanism_param*.

AES_CTR = 4230

AES_CTS = 4233

AES_MAC = 4227

Note: This is the default for signing/verification with *KeyType.AES*.

AES_MAC_GENERAL = 4228

AES_CMAC = 4234

AES_CMAC_GENERAL = 4235

BLOWFISH_KEY_GEN = 4240

BLOWFISH_CBC = 4241

BLOWFISH_CBC_PAD = 4244

TWOFISH_KEY_GEN = 4242

TWOFISH_CBC = 4243

TWOFISH_CBC_PAD = 4245

AES_GCM = 4231

AES_CCM = 4232

AES_XCBC_MAC = 4236

AES_XCBC_MAC_96 = 4237

AES_GMAC = 4238

AES_OFB = 8452

AES_CFB64 = 8453

AES_CFB8 = 8454

AES_CFB128 = 8455

AES_CFB1 = 8456

AES_KEY_WRAP = 8457

AES_KEY_WRAP_PAD = 8458

DES_ECB_ENCRYPT_DATA = 4352

DES_CBC_ENCRYPT_DATA = 4353

DES3_ECB_ENCRYPT_DATA = 4354

DES3_CBC_ENCRYPT_DATA = 4355

AES_ECB_ENCRYPT_DATA = 4356

AES_CBC_ENCRYPT_DATA = 4357

GOSTR3410_KEY_PAIR_GEN = 4608

GOSTR3410 = 4609

GOSTR3410_WITH_GOSTR3411 = 4610

GOSTR3410_KEY_WRAP = 4611

GOSTR3410_DERIVE = 4612

GOSTR3411 = 4624

GOSTR3411_HMAC = 4625

GOST28147_KEY_GEN = 4640

GOST28147_ECB = 4641

> **Warning:** Identical blocks will encipher to the same result.

GOST28147 = 4642

GOST28147_MAC = 4643

GOST28147_KEY_WRAP = 4644

DSA_PARAMETER_GEN = 8192

> **Note:** Default mechanism for generating *KeyType.DSA* domain parameters.

DH_PKCS_PARAMETER_GEN = 8193

> **Note:** Default mechanism for generating *KeyType.DH* domain parameters.

This is the mechanism defined in PKCS #3.

X9_42_DH_PARAMETER_GEN = 8194

> **Note:** Default mechanism for generating *KeyType.X9_42_DH* domain parameters (X9.42 DH).

EDDSA = 4183

EC_EDWARDS_KEY_PAIR_GEN = 4181

class pkcs11.mechanisms.**KDF**
Key Derivation Functions.

NULL = 1

SHA1 = 2

SHA1_ASN1 = 3

SHA1_CONCATENATE = 4

SHA224 = 5

SHA256 = 6

SHA384 = 7

SHA512 = 8

CPDIVERSIFY = 9

class pkcs11.mechanisms.**MGF**
RSA PKCS #1 Mask Generation Functions.

SHA1 = 1

SHA256 = 2

SHA384 = 3

SHA512 = 4

SHA224 = 5

class pkcs11.**MechanismInfo**
Information about a mechanism.

See *pkcs11.Slot.get_mechanism_info()*.

slot = None
pkcs11.Slot this information is for.

mechanism = None
pkcs11.mechanisms.Mechanism this information is for.

min_key_length = None
Minimum key length in bits (int).

max_key_length = None
Maximum key length in bits (int).

flags = None
Mechanism capabilities (*pkcs11.constants.MechanismFlag*).

5.2.4 Exceptions

PKCS#11 return codes are exposed as Python exceptions inheriting from *PKCS11Error*.

exception pkcs11.exceptions.**PKCS11Error**
Base exception for all PKCS#11 exceptions.

exception pkcs11.exceptions.**AlreadyInitialized**
pkcs11 was already initialized with another library.

exception pkcs11.exceptions.**AnotherUserAlreadyLoggedIn**

exception pkcs11.exceptions.**AttributeTypeInvalid**

exception pkcs11.exceptions.**AttributeValueInvalid**

exception pkcs11.exceptions.**AttributeReadOnly**
An attempt was made to set a value for an attribute which may not be set by the application, or which may not be modified by the application.

exception pkcs11.exceptions.**AttributeSensitive**
An attempt was made to obtain the value of an attribute of an object which cannot be satisfied because the object is either sensitive or un-extractable.

exception pkcs11.exceptions.**ArgumentsBad**
Bad arguments were passed into PKCS#11.

This can indicate missing parameters to a mechanism or some other issue. Consult your PKCS#11 vendor documentation.

exception pkcs11.exceptions.**DataInvalid**
The plaintext input data to a cryptographic operation is invalid.

exception pkcs11.exceptions.**DataLenRange**
The plaintext input data to a cryptographic operation has a bad length. Depending on the operation's mechanism, this could mean that the plaintext data is too short, too long, or is not a multiple of some particular block size.

exception pkcs11.exceptions.**DomainParamsInvalid**
Invalid or unsupported domain parameters were supplied to the function. Which representation methods of domain parameters are supported by a given mechanism can vary from token to token.

exception pkcs11.exceptions.**DeviceError**

exception pkcs11.exceptions.**DeviceMemory**
The token does not have sufficient memory to perform the requested function.

exception pkcs11.exceptions.**DeviceRemoved**
The token was removed from its slot during the execution of the function.

exception pkcs11.exceptions.**EncryptedDataInvalid**
The encrypted input to a decryption operation has been determined to be invalid ciphertext.

exception pkcs11.exceptions.**EncryptedDataLenRange**
The ciphertext input to a decryption operation has been determined to be invalid ciphertext solely on the basis of its length. Depending on the operation's mechanism, this could mean that the ciphertext is too short, too long, or is not a multiple of some particular block size.

exception pkcs11.exceptions.**ExceededMaxIterations**
An iterative algorithm (for key pair generation, domain parameter generation etc.) failed because we have exceeded the maximum number of iterations.

exception pkcs11.exceptions.**FunctionCancelled**

exception pkcs11.exceptions.**FunctionFailed**

exception pkcs11.exceptions.**FunctionRejected**

exception pkcs11.exceptions.**FunctionNotSupported**

exception pkcs11.exceptions.**KeyHandleInvalid**

exception pkcs11.exceptions.**KeyIndigestible**

exception pkcs11.exceptions.**KeyNeeded**

exception pkcs11.exceptions.**KeyNotNeeded**

exception pkcs11.exceptions.**KeyNotWrappable**

exception pkcs11.exceptions.**KeySizeRange**

exception pkcs11.exceptions.**KeyTypeInconsistent**

exception pkcs11.exceptions.**KeyUnextractable**

exception pkcs11.exceptions.**GeneralError**

In unusual (and extremely unpleasant!) circumstances, a function can fail with the return value CKR_GENERAL_ERROR. When this happens, the token and/or host computer may be in an inconsistent state, and the goals of the function may have been partially achieved.

exception pkcs11.exceptions.**HostMemory**

The computer that the Cryptoki library is running on has insufficient memory to perform the requested function.

exception pkcs11.exceptions.**MechanismInvalid**

Mechanism can not be used with requested operation.

exception pkcs11.exceptions.**MechanismParamInvalid**

exception pkcs11.exceptions.**MultipleObjectsReturned**

Multiple objects matched the search parameters.

exception pkcs11.exceptions.**MultipleTokensReturned**

Multiple tokens matched the search parameters.

exception pkcs11.exceptions.**NoEvent**

No slot events were detected.

exception pkcs11.exceptions.**NoSuchKey**

No key matching the parameters was found.

exception pkcs11.exceptions.**NoSuchToken**

No token matching the parameters was found.

exception pkcs11.exceptions.**ObjectHandleInvalid**

exception pkcs11.exceptions.**OperationActive**

There is already an active operation (or combination of active operations) which prevents Cryptoki from activating the specified operation. For example, an active object-searching operation would prevent Cryptoki from activating an encryption operation with C_EncryptInit. Or, an active digesting operation and an active encryption operation would prevent Cryptoki from activating a signature operation. Or, on a token which doesn't support simultaneous dual cryptographic operations in a session (see the description of the CKF_DUAL_CRYPTO_OPERATIONS flag in the CK_TOKEN_INFO structure), an active signature operation would prevent Cryptoki from activating an encryption operation.

exception pkcs11.exceptions.**OperationNotInitialized**

exception pkcs11.exceptions.**PinExpired**

exception pkcs11.exceptions.**PinIncorrect**

exception pkcs11.exceptions.**PinInvalid**

exception pkcs11.exceptions.**PinLenRange**
The specified PIN is too long or too short.

exception pkcs11.exceptions.**PinLocked**

exception pkcs11.exceptions.**PinTooWeak**

exception pkcs11.exceptions.**PublicKeyInvalid**

exception pkcs11.exceptions.**RandomNoRNG**

exception pkcs11.exceptions.**RandomSeedNotSupported**

exception pkcs11.exceptions.**SessionClosed**
The session was closed during the execution of the function.

exception pkcs11.exceptions.**SessionCount**
An attempt to open a session which does not succeed because there are too many existing sessions.

exception pkcs11.exceptions.**SessionExists**

exception pkcs11.exceptions.**SessionHandleInvalid**
The session handle was invalid. This is usually caused by using an old session object that is not known to PKCS#11.

exception pkcs11.exceptions.**SessionReadOnly**
Attempted to write to a read-only session.

exception pkcs11.exceptions.**SessionReadOnlyExists**

exception pkcs11.exceptions.**SessionReadWriteSOExists**
If the application calling `Token.open()` already has a R/W SO session open with the token, then any attempt to open a R/O session with the token fails with this exception.

exception pkcs11.exceptions.**SignatureLenRange**

exception pkcs11.exceptions.**SignatureInvalid**

exception pkcs11.exceptions.**SlotIDInvalid**

exception pkcs11.exceptions.**TemplateIncomplete**
Required attributes to create the object were missing.

exception pkcs11.exceptions.**TemplateInconsistent**
Template values (including vendor defaults) are contradictory.

exception pkcs11.exceptions.**TokenNotPresent**
The token was not present in its slot at the time that the function was invoked.

exception pkcs11.exceptions.**TokenNotRecognised**

exception pkcs11.exceptions.**TokenWriteProtected**

exception pkcs11.exceptions.**UnwrappingKeyHandleInvalid**

exception pkcs11.exceptions.**UnwrappingKeySizeRange**

exception pkcs11.exceptions.**UnwrappingKeyTypeInconsistent**

exception pkcs11.exceptions.**UserAlreadyLoggedIn**

exception pkcs11.exceptions.**UserNotLoggedIn**

exception pkcs11.exceptions.**UserPinNotInitialized**

exception pkcs11.exceptions.**UserTooManyTypes**
An attempt was made to have more distinct users simultaneously logged into the token than the token and/or library permits. For example, if some application has an open SO session, and another application attempts to log the normal user into a session, the attempt may return this error. It is not required to, however. Only if the simultaneous distinct users cannot be supported does C_Login have to return this value. Note that this error code generalizes to true multi-user tokens.

exception pkcs11.exceptions.**WrappedKeyInvalid**

exception pkcs11.exceptions.**WrappedKeyLenRange**

exception pkcs11.exceptions.**WrappingKeyHandleInvalid**

exception pkcs11.exceptions.**WrappingKeySizeRange**

exception pkcs11.exceptions.**WrappingKeyTypeInconsistent**

5.2.5 Utilities

General Utilities

pkcs11.util.**biginteger**(*value*)
Returns a PKCS#11 biginteger bytestream from a Python integer or similar type (e.g. `asn1crypto.core.Integer`).

Parameters **value** (*int*) – Value

Return type bytes

RSA Key Utilities

Key handling utilities for RSA keys (PKCS#1).

pkcs11.util.rsa.**decode_rsa_private_key**(*der*, *capabilities=None*)
Decode a RFC2437 (PKCS#1) DER-encoded RSA private key into a dictionary of attributes able to be passed to *pkcs11.Session.create_object()*.

Parameters

- **der** (*bytes*) – DER-encoded key
- **capabilities** (MechanismFlag) – Optional key capabilities

Return type dict(*Attribute*,*)

pkcs11.util.rsa.**decode_rsa_public_key**(*der*, *capabilities=None*)
Decode a RFC2437 (PKCS#1) DER-encoded RSA public key into a dictionary of attributes able to be passed to *pkcs11.Session.create_object()*.

Parameters

- **der** (*bytes*) – DER-encoded key
- **capabilities** (MechanismFlag) – Optional key capabilities

Return type dict(*Attribute*,*)

pkcs11.util.rsa.**encode_rsa_public_key**(*key*)
Encode an RSA public key into PKCS#1 DER-encoded format.

Parameters **key** (PublicKey) – RSA public key

Return type bytes

DSA Key Utilities

Key handling utilities for DSA keys, domain parameters and signatures..

`pkcs11.util.dsa.`**`decode_dsa_domain_parameters`**`(`*der*`)`
Decode RFC 3279 DER-encoded Dss-Params.

Parameters **der** (*bytes*) – DER-encoded parameters

Return type dict(*Attribute*,*)

`pkcs11.util.dsa.`**`encode_dsa_domain_parameters`**`(`*obj*`)`
Encode RFC 3279 DER-encoded Dss-Params.

Parameters **obj** (DomainParameters) – domain parameters

Return type bytes

`pkcs11.util.dsa.`**`encode_dsa_public_key`**`(`*key*`)`
Encode DSA public key into RFC 3279 DER-encoded format.

Parameters **key** (PublicKey) – public key

Return type bytes

`pkcs11.util.dsa.`**`decode_dsa_public_key`**`(`*der*`)`
Decode a DSA public key from RFC 3279 DER-encoded format.

Returns a *biginteger* encoded as bytes.

Parameters **der** (*bytes*) – DER-encoded public key

Return type bytes

`pkcs11.util.dsa.`**`encode_dsa_signature`**`(`*signature*`)`
Encode a signature (generated by *pkcs11.SignMixin.sign()*) into DER-encoded ASN.1 (Dss_Sig_Value) format.

Parameters **signature** (*bytes*) – signature as bytes

Return type bytes

`pkcs11.util.dsa.`**`decode_dsa_signature`**`(`*der*`)`
Decode a DER-encoded ASN.1 (Dss_Sig_Value) signature (as generated by OpenSSL/X.509) into PKCS #11 format.

Parameters **der** (*bytes*) – DER-encoded signature

Rtype bytes

DH Key Utilities

Key handling utilities for Diffie-Hellman keys.

`pkcs11.util.dh.`**`decode_dh_domain_parameters`**`(`*der*`)`
Decode DER-encoded Diffie-Hellman domain parameters.

Parameters **der** (*bytes*) – DER-encoded parameters

Return type dict(*Attribute*,*)

`pkcs11.util.dh.`**`encode_dh_domain_parameters`**(*obj*)

Encode DH domain parameters into DER-encoded format.

Calculates the subprime if it isn't available.

Parameters **`obj`** (`DomainParameters`) – domain parameters

Return type bytes

`pkcs11.util.dh.`**`encode_dh_public_key`**(*key*)

Encode DH public key into RFC 3279 DER-encoded format.

Parameters **`key`** (`PublicKey`) – public key

Return type bytes

`pkcs11.util.dh.`**`decode_dh_public_key`**(*der*)

Decode a DH public key from RFC 3279 DER-encoded format.

Returns a *biginteger* encoded as bytes.

Parameters **`der`** (*`bytes`*) – DER-encoded public key

Return type bytes

EC Key Utilities

Key handling utilities for EC keys (ANSI X.62/RFC3279), domain parameter and signatures.

`pkcs11.util.ec.`**`encode_named_curve_parameters`**(*oid*)

Return DER-encoded ANSI X.62 EC parameters for a named curve.

Curve names are given by object identifier or common name. Names come from asn1crypto.

Parameters **`oid`** (*`str`*) – OID or named curve

Return type bytes

`pkcs11.util.ec.`**`decode_ec_public_key`**(*der*, *encode_ec_point=True*)

Decode a DER-encoded EC public key as stored by OpenSSL into a dictionary of attributes able to be passed to *`pkcs11.Session.create_object()`*.

Note: encode_ec_point

For use as an attribute *EC_POINT* should be DER-encoded (True).

For key derivation implementations can vary. Since v2.30 the specification says implementations MUST accept a raw *EC_POINT* for ECDH (False), however not all implementations follow this yet.

Parameters

- **`der`** (*`bytes`*) – DER-encoded key
- **`encode_ec_point`** – See text.

Return type dict(*Attribute*,*)

`pkcs11.util.ec.`**`decode_ec_private_key`**(*der*)

Decode a DER-encoded EC private key as stored by OpenSSL into a dictionary of attributes able to be passed to *`pkcs11.Session.create_object()`*.

Parameters **`der`** (*`bytes`*) – DER-encoded key

Return type dict(*Attribute*,*)

`pkcs11.util.ec.`**`encode_ec_public_key`**(*key*)

Encode a DER-encoded EC public key as stored by OpenSSL.

Parameters **`key`** (`PublicKey`) – EC public key

Return type bytes

`pkcs11.util.ec.`**`encode_ecdsa_signature`**(*signature*)

Encode a signature (generated by *`pkcs11.SignMixin.sign()`*) into DER-encoded ASN.1 (ECDSA_Sig_Value) format.

Parameters **`signature`** (*`bytes`*) – signature as bytes

Return type bytes

`pkcs11.util.ec.`**`decode_ecdsa_signature`**(*der*)

Decode a DER-encoded ASN.1 (ECDSA_Sig_Value) signature (as generated by OpenSSL/X.509) into PKCS #11 format.

Parameters **`der`** (*`bytes`*) – DER-encoded signature

Rtype bytes

X.509 Certificate Utilities

Certificate handling utilities for X.509 (SSL) certificates.

`pkcs11.util.x509.`**`decode_x509_public_key`**(*der*)

Decode a DER-encoded X.509 certificate's public key into a set of attributes able to be passed to *`pkcs11.Session.create_object()`*.

For PEM-encoded certificates, use `asn1crypto.pem.unarmor()`.

Warning: Does not verify certificate.

Parameters **`der`** (*`bytes`*) – DER-encoded certificate

Return type dict(*Attribute*,*)

`pkcs11.util.x509.`**`decode_x509_certificate`**(*der*, *extended_set=False*)

Decode a DER-encoded X.509 certificate into a dictionary of attributes able to be passed to *`pkcs11.Session.create_object()`*.

Optionally pass *extended_set* to include additional attributes: start date, end date and key identifiers.

For PEM-encoded certificates, use `asn1crypto.pem.unarmor()`.

Warning: Does not verify certificate.

Parameters

- **`der`** (*`bytes`*) – DER-encoded certificate
- **`extended_set`** – decodes more metadata about the certificate

Return type dict(*Attribute*,*)

5.3 Concurrency

PKCS#11 is able to be accessed from multiple threads. The specification recommends setting a flag to enable access from multiple threads, however due to the existence of the global interpreter lock preventing concurrent execution of Python threads, you will not be preempted inside a single PKCS#11 call and so the flag has not been set to maximise compatibility with PKCS#11 implementations.

Most of the calls exposed in our API make a single call into PKCS#11, however, multi-step calls, such as searching for objects, encryption, decryption, etc. can be preempted as control is returned to the interpreter (e.g. by generators). The *pkcs11.Session* class includes a reentrant lock (`threading.RLock`) to control access to these multi-step operations, and prevent threads from interfering with each other.

> **Warning:** Libraries that monkeypatch Python, such as *gevent*, may be supported, but are not currently being tested.

The lock is not released until the iterator is consumed (or garbage collected). However, if you do not consume the iterator, you will never complete the action and further actions will raise *pkcs11.exceptions.OperationActive* (cancelling iterators is not currently supported).

5.3.1 Reentrant Sessions

Thread safety aside, a number of PKCS#11 libraries do not support the same token being logged in from simultaneous sessions (within the same process), and so it can be advantageous to use a single session across multiple threads. Sessions can often live for a very long time, but failing to close a session may leak resources into your memory space, HSM daemon or HSM hardware.

A simple reference counting reentrant session object can be used.

```
import logging
import threading

import pkcs11


LOCK = threading.Lock()
LIB = pkcs11.lib(settings.PKCS11_MODULE)


class Session(object):
    """Reentrant session wrapper."""

    session = None
    refcount = 0

    @classmethod
    def acquire(cls):
        with LOCK:
            if cls.refcount == 0:
                token = LIB.get_token(token_label=settings.PKCS11_TOKEN)
                cls.session = token.open(user_pin=settings.PKCS11_TOKEN_PASSPHRASE)

            cls.refcount += 1
            return cls.session
```

(continues on next page)

(continued from previous page)

```
    @classmethod
    def release(cls):
        with LOCK:
            cls.refcount -= 1

            if cls.refcount == 0:
                cls.session.close()
                cls.session = None

    def __enter__(self):
        return self.acquire()

    def __exit__(self, type_, value, traceback):
        self.release()
```

The multi-step locking primitives in the `pkcs11.Session` should allow you to operate safely.

5.4 Using with SmartCard-HSM (Nitrokey HSM)

Support for the SmartCard-HSM and Nitrokey HSM is provided through the OpenSC project.

The device is not a cryptographic accelerator. Only key generation and the private key operations (sign and decrypt) are supported. Public key operations should be done by extracting the public key and working on the computer.

The following mechanisms are available:

Cipher	Capabilities	Variants
RSA (v1.5/X.509)	Decrypt, Verify, Sign	MD5, SHA1, SHA256, SHA384, SHA512
ECDSA	Sign	SHA1
ECDH	Derive	Cofactor Derive

Session lifetime objects are not supported and the value of `pkcs11.constants.Attribute.TOKEN` and the *store* keyword argument are ignored. All objects will be stored to the device.

The following named curves are supported:

- secp192r1 (aka prime192v1)
- secp256r1 (aka prime256v1)
- brainpoolP192r1
- brainpoolP224r1
- brainpoolP256r1
- brainpoolP320r1
- secp192k1
- secp256k1 (the Bitcoin curve)

More information is available in the Nitrokey FAQ.

5.4.1 Getting Started

Initialize the device with *sc-hsm-tool*, e.g.

```
sc-hsm-tool --initialize --so-pin 3537363231383830 --pin 648219 --label "Nitrokey"
```

See the documentation for more information on the parameters.

The OpenSC PKCS #11 module is *opensc-pkcs11.so*.

5.4.2 Generating Keys

RSA

```
import pkcs11

with token.open(user_pin='1234', rw=True) as session:
    pub, priv = session.generate_keypair(pkcs11.KeyType.RSA, 2048,
                                         store=True,
                                         label="My RSA Keypair")
```

EC

```
with token.open(user_pin='1234', rw=True) as session:
    ecparams = session.create_domain_parameters(
        pkcs11.KeyType.EC, {
            pkcs11.Attribute.EC_PARAMS: pkcs11.util.ec.encode_named_curve_parameters(
↪'secp256r1'),
        }, local=True)

    pub, priv = ecparams.generate_keypair(store=True,
                                          label="My EC Keypair")
```

5.4.3 Exporting Public Keys for External Use

While we don't want our private keys to leave the boundary of our HSM, we can extract the public keys for use with a cryptographic library of our choosing. *Importing/Exporting Keys* has more information on functions for exporting keys.

RSA

PyCrypto example:

```
from pkcs11 import KeyType, ObjectClass, Mechanism
from pkcs11.util.rsa import encode_rsa_public_key

from Crypto.PublicKey import RSA
from Crypto.Cipher import PKCS1_v1_5

# Extract public key
key = session.get_key(key_type=KeyType.RSA,
```

(continues on next page)

(continued from previous page)

```
                              object_class=ObjectClass.PUBLIC_KEY)
key = RSA.importKey(encode_rsa_public_key(key))

# Encryption on the local machine
cipher = PKCS1_v1_5.new(key)
crypttext = cipher.encrypt(b'Data to encrypt')

# Decryption in the HSM
priv = self.session.get_key(key_type=KeyType.RSA,
                            object_class=ObjectClass.PRIVATE_KEY)

plaintext = priv.decrypt(crypttext, mechanism=Mechanism.RSA_PKCS)
```

ECDSA

oscrypto example:

```
from pkcs11 import KeyType, ObjectClass, Mechanism
from pkcs11.util.ec import encode_ec_public_key, encode_ecdsa_signature

from oscrypto.asymmetric import load_public_key, ecdsa_verify

# Sign data in the HSM
priv = self.session.get_key(key_type=KeyType.EC,
                            object_class=ObjectClass.PRIVATE_KEY)
signature = priv.sign(b'Data to sign', mechanism=Mechanism.ECDSA_SHA1)
# Encode as ASN.1 for interchange
signature = encode_ecdsa_signature(signature)

# Extract the public key
pub = self.session.get_key(key_type=KeyType.EC,
                           object_class=ObjectClass.PUBLIC_KEY)

# Verify the signature on the local machine
key = load_public_key(encode_ec_public_key(pub))
ecdsa_verify(key, signature, b'Data to sign', 'sha1')
```

ECDH

Smartcard-HSM can generate a shared key via ECDH key exchange.

Warning: Where possible, e.g. over networks, you should use ephemeral keys, to allow for perfect forward secrecy. Smartcard HSM's ECDH is only useful when need to repeatedly retrieve the same shared secret, e.g. encrypting files in a hybrid cryptosystem.

cryptography example:

```
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives.asymmetric import ec
from cryptography.hazmat.primitives.serialization import \
    Encoding, PublicFormat, load_der_public_key
```

(continues on next page)

(continued from previous page)

```
# Retrieve our keypair, with our public key encoded for interchange
alice_priv = self.session.get_key(key_type=KeyType.EC,
                                  object_class=ObjectClass.PRIVATE_KEY)
alice_pub = self.session.get_key(key_type=KeyType.EC,
                                 object_class=ObjectClass.PUBLIC_KEY)
alice_pub = encode_ec_public_key(alice_pub)

# Bob generates a keypair, with their public key encoded for
# interchange
bob_priv = ec.generate_private_key(ec.SECP256R1,
                                   default_backend())
bob_pub = bob_priv.public_key().public_bytes(
    Encoding.DER,
    PublicFormat.SubjectPublicKeyInfo,
)

# Bob converts Alice's key to internal format and generates their
# shared key
bob_shared_key = bob_priv.exchange(
    ec.ECDH(),
    load_der_public_key(alice_pub, default_backend()),
)

key = alice_priv.derive_key(
    KeyType.GENERIC_SECRET, 256,
    mechanism_param=(
        KDF.NULL, None,
        # SmartcardHSM doesn't accept DER-encoded EC_POINTs for derivation
        decode_ec_public_key(bob_pub, encode_ec_point=False)
        [Attribute.EC_POINT],
    ),
)
alice_shared_key = key[Attribute.VALUE]
```

When decoding the other user's *EC_POINT* for passing into the key derivation the standard says to pass a raw octet string (set *encode_ec_point* to False), however some PKCS #11 implementations require a DER-encoded octet string (i.e. the format of the *pkcs11.constants.Attribute.EC_POINT* attribute).

5.4.4 Encrypting Files

The device only supports asymmetric mechanisms. To do file encryption, you will need to generate AES keys locally, which you can encrypt with your RSA public key (this is how the Nitrokey storage key works); or by using ECDH to generate a shared secret from a locally generated public key.

5.4.5 Debugging

The parameter *OPENSC_DEBUG* will enable debugging of the OpenSC driver. A higher number indicates more verbosity.

5.4.6 Thanks

Thanks to Nitrokey for their support of open software and sending a Nitrokey HSM to test with *python-pkcs11*.

CHAPTER

SIX

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

p

INDEX

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

X