CryptoJWT Documentation

Release 0.4

Roland Hedberg

Jun 22, 2022

Contents

CryptoJWT is supposed to provide you (a Python programmer) with all you need, to deal with what is described in RFC 7515-7519.

CHAPTER 1

How to deal with Cryptographic keys

Absolutely vital to doing cryptography is to be able to handle keys. This document will show you have to accomplish this with this package.

CryptoJWT deals with keys by defining 5 ‘layers’.

1. At the bottom we have the keys as used by the cryptographic package (in this case cryptography) .
2. Above that we have something we call a JSON Web Key. The base class here is `cryptojwt.jwk.JWK`. This class can import keys in a number of formats and can export a key as a JWK.
3. A `cryptojwt.key_bundle.KeyBundle` keeps track of a set of keys that has the same origin. Like being part of a JWKS.
4. A `cryptojwt.key_issuer.KeyIssuer` keeps keys per owners/issuers.
5. A `cryptojwt.key_jar.KeyJar` lastly is there to keep the keys sorted by their owners/issuers.

I will not describe how to deal with keys in layer 1, that is done best by cryptography. So, I’ll start at layer 2.

1.1 JSON Web Key (JWK)

Let us start with you not having any key at all and you want to create a signed JSON Web Token (JWS). What to do ?

1.1.1 Staring with no key

Well if you know what kind of key you want, and if it is a asymmetric key you want, you can use one of the provided factory methods.

RSA `cryptojwt.jwk.rsa.new_rsa_key()`

Elliptic Curve: `cryptojwt.jwk.ec.new_ec_key()`

As an example:

```
>>> from cryptojwt.jwk.rsa import new_rsa_key
>>> rsa_key = new_rsa_key()
>>> type(rsa_key)
<class 'cryptojwt.jwk.rsa.RSAKey'>
```

or if you want an elliptic curve key:

```
>>> from cryptojwt.jwk.ec import new_ec_key
>>> ec_key = new_ec_key('P-256')
>>> type(ec_key)
<class 'cryptojwt.jwk.ec.ECKey'>
>>> ec_key.has_private_key()
True
```

If you want a symmetric key you only need some sort of "secure random" mechanism. You can use this to acquire a byte array of the appropriate length (e.g. 32 bytes for AES256), which can be used as a key.

1.1.2 When you have a key in a file on your hard drive

These functions will return keys as cryptography class instances.

PEM encoded private key `cryptojwt.jwk.x509.import_private_key_from_pem_file()`

PEM encoded public key `cryptojwt.jwk.x509.import_public_key_from_pem_file()`

JWK `cryptojwt.jwk.jwk.import_jwk()`

Here are some examples:

```
>>> from cryptojwt.jwk.rsa import RSAKey
>>> from cryptojwt.jwk.rsa import import_private_rsa_key_from_file
>>> _key = import_private_rsa_key_from_file('rsa-key.pem')
>>> rsa_key = RSAKey(priv_key=_key)
>>> rsa_key.has_private_key()
True
```

If you have a PEM encoded X.509 certificate you may want to grab the public RSA key from you could do like this:

```
>>> from cryptojwt.jwk.x509 import import_public_key_from_cert_file
>>> from cryptojwt.jwk.rsa import RSAKey
>>> _key = import_public_key_from_cert_file('cert.pem')
>>> rsa_key = RSAKey(pub_key=_key)
>>> rsa_key.has_private_key()
False
>>> type(rsa_key.public_key())
<class 'cryptography.hazmat.backends.openssl.rsa._RSAPublicKey'>
```

If you are dealing with Elliptic Curve keys the equivalent operations would be:

```
>>> from cryptojwt.jwk.ec import ECKey
>>> from cryptojwt.jwk.ec import import_private_ec_key_from_file
>>> _key = import_private_ec_key_from_file('ec-private.pem')
>>> ec_key = ECKey(priv_key=_key)
>>> ec_key.has_private_key()
True
```

and

```
>>> from cryptojwt.jwk.x509 import import_public_key_from_cert_file
>>> from cryptojwt.jwk.ec import ECKey
>>> _key = import_public_key_from_cert_file('ec_certificate.pem')
>>> ec_key = ECKey(pub_key=_key)
>>> ec_key.has_private_key()
False
>>> type(ec_key.public_key())
<class 'cryptography.hazmat.backends.openssl.ec._EllipticCurvePublicKey'>
```

To import a JWK encoded key from a file, you do

```
>>> from cryptojwt.jwk.jwk import import_jwk
>>> from cryptojwt.jwk.rsa import RSAKey
>>> _jwk = import_jwk('jwk.json')
>>> isinstance(_jwk, RSAKey)
True
```

1.1.3 Exporting keys

When it comes to exporting keys, a `cryptojwt.jwk.JWK` instance only knows how to serialize into the format described in JWK.

```
>>> from cryptojwt.jwk.ec import ECKey
>>> from cryptojwt.jwk.ec import import_private_ec_key_from_file
>>> import json
>>> _key = import_private_ec_key_from_file('ec-private.pem')
>>> ec_key = ECKey(priv_key=_key)
>>> _jwk_keys = list(ec_key.serialize(private=True).keys())
>>> _jwk_keys.sort()
>>> print(_jwk_keys)
['crv', 'd', 'kty', 'x', 'y']
>>> _jwk_keys = list(ec_key.serialize(private=False).keys())
>>> _jwk_keys.sort()
>>> print(_jwk_keys)
['crv', 'kty', 'x', 'y']
```

What you get when doing it like above is a representation of the public key. You can also get the values for the private key like this:

```
>>> from cryptojwt.jwk.rsa import RSAKey
>>> from cryptojwt.jwk.rsa import import_private_rsa_key_from_file
>>> _key = import_private_rsa_key_from_file('rsa-key.pem')
>>> rsa_key = RSAKey(priv_key=_key)
>>> _jwk_keys = list(rsa_key.serialize(private=True).keys())
>>> _jwk_keys.sort()
>>> print(_jwk_keys)
['d', 'e', 'kty', 'n', 'p', 'q']
```

And you can of course create a key from a JWK representation:

```
>>> from cryptojwt.jwk.rsa import new_rsa_key
>>> from cryptojwt.jwk.jwk import key_from_jwk_dict
>>> rsa_key = new_rsa_key()
>>> jwk = rsa_key.serialize(private=True)
>>> _key = key_from_jwk_dict(jwk)
```

(continues on next page)

(continued from previous page)

```
>>> type(_key)
<class 'cryptojwt.jwk.rsa.RSAKey'>
>>> _key.has_private_key()
True
```

1.2 Key bundle

As mentioned above a key bundle is used to manage keys that have a common origin.

You can initiate a key bundle in several ways. You can use all the import variants we described above and then add the resulting key to a key bundle:

```
>>> from cryptojwt.jwk.ec import ECKey
>>> from cryptojwt.key_bundle import KeyBundle
>>> ec_key = ECKey().load('ec-private.pem')
>>> kb = KeyBundle()
>>> rsa_key = RSAKey().load('rsa-key.pem')
>>> kb.set([ec_key, rsa_key])
>>> len(kb)
2
```

Note that this will get you a JWKS (which is not the same as a JWK) representing the public keys.

As an example of the special functionality of `cryptojwt.key_bundle.KeyBundle` assume you have imported a file containing a JWKS with a couple of keys into a key bundle and then some time later a key is added to the file.

First import the file with one key:

```
>>> from cryptojwt.key_bundle import KeyBundle
>>> fname="private_jwks.json"
>>> kb = KeyBundle(source="file://{}".format(fname), fileformat='jwks')
>>> len(kb)
3
```

Now if we add one key to the file and then some time later we ask for the keys in the key bundle:

```
>>> _keys = kb.keys()
>>> len(_keys)
4
```

It turns out the key bundle now contains 4 keys. All of them coming from the file. When you use the method *keys* the bundle is automatically updated with the new information in the file.

If you want to be explicit you can use the method *update*.

If the change is that one key is removed then something else happens. Assume we add one key and remove one of the keys that was there before. The file still contains 3 keys, and you might expect the key bundle to do the same:

```
>>> _keys = kb.keys()
>>> len(_keys)
4
```

What ??? The key that was removed has not disappeared from the key bundle, but it is marked as *inactive*. Which means that it should not be used for signing and encryption but can be used for decryption and signature verification. This is to aid when someone does key rotation

```
>>> len(kb.get('rsa'))
1
>>> len(kb.get('rsa', only_active=False))
2
```

The last thing you need to know when it comes to importing keys into a key bundle is how to keep them updated. If the source of the keys are a file you fetched over the net, or a file in your own file system, that file may be updated at any time. The same goes for a local file which you may or may not be in control of. To go through and update all the keys in a key bundle you can use the method *update*:

```
>>> from cryptojwt.key_bundle import KeyBundle
>>> kb = KeyBundle(source="https://www.googleapis.com/oauth2/v3/certs",
...   fileformat='jwks')
```

And sometime later

```
>>> kb.update()
True
```

Now you know how to get keys into a key bundle next step is to find keys. There are two methods *get* and *get_key_with_kid*. You use *get* when you just want a key you can use

```
>>> kb = KeyBundle(source="https://www.googleapis.com/oauth2/v3/certs",
...   fileformat='jwks')
>>> rsa_keys = kb.get(typ='rsa')
>>> sig_keys = [k.appropriate_for('verify') for k in rsa_keys]
>>> sig_keys is not []
True
```

And *get_key_with_kid* when you want a specific key:

```
>>> kb = KeyBundle(source="https://www.googleapis.com/oauth2/v3/certs",
...   fileformat='jwks')
>>> key_ids = kb.kids()
>>> key = kb.get_key_with_kid(key_ids[0])
>>> type(key)
<class 'cryptojwt.jwk.rsa.RSAKey'>
```

And of course you may want to export a JWKS

```
>>> from cryptojwt.key_bundle import KeyBundle
>>> import json
>>> fname="private_jwks.json"
>>> kb = KeyBundle(source="file://{}".format(fname), fileformat='jwks')
>>> _jwks = kb.jwks(private=True)
>>> kb2 = KeyBundle()
>>> kb2.do_keys(json.loads(_jwks)["keys"])
>>> kb.difference(kb2)
[]
>>> kb2.difference(kb)
[]
```

1.3 Key Issuer

All the keys that you are dealing with will be owned by/connected to some entity. The `cryptojwt.key_issuer.KeyIssuer` class helps you keep them together. A key issuer instance contains one or more key bundles. To borrow from earlier examples:

```
>>> from cryptojwt.key_issuer import KeyIssuer
>>> issuer = KeyIssuer(name="https://example.com")
>>> issuer.import_jwks_from_file('ec-p256.json')
>>> issuer.import_jwks_from_file('rsa_jwks.json')
>>> len(issuer)
2
```

issuer contains two key bundles each with one key each.

In most cases it does not matter which key bundle a certain key is placed in you just want a key. For this you have the *get* method which takes a number of arguments

```
>>> from cryptojwt.key_bundle import keybundle_from_local_file
>>> from cryptojwt.key_issuer import KeyIssuer
>>> kb = keybundle_from_local_file('public_jwks.json', typ='jwks')
>>> issuer = KeyIssuer(name="https://example.com")
>>> issuer.add_kb(kb)
>>> keys = issuer.get(key_use='sig', key_type="ec", alg="ES256")
>>> len(keys)
1
>>> keys = issuer.get(key_use='sig', key_type="ec", crv="P-256")
>>> len(keys)
1
>>> keys = issuer.get(key_use='sig', key_type="ec")
>>> len(keys)
2
```

1.4 Key Jar

A key jar keeps keys sorted by owner/issuer. The keys in a `cryptojwt.key_jar.KeyJar` instance are contained in `cryptojwt.key_issuer.KeyIssuer` instances.

Creating a key jar with your own newly minted keys you would do:

```
>>> from cryptojwt.key_jar import build_keyjar
>>> key_specs = [{"type": "RSA", "use": ["enc", "sig"]},{"type": "EC", "crv": "P-256",
↪ "use": ["sig"]}]
>>> key_jar = build_keyjar(key_specs)
>>> len(key_jar[''].all_keys())
3
```

Note that the default issuer ID is the empty string ‘’. **Note** also that different RSA keys are minted for signing and for encryption.

You can also use `cryptojwt.keyjar.init_key_jar()` which will load keys from disk if they are there and if not mint new accoring to a provided specification.:

```
>>> from cryptojwt.key_jar import init_key_jar
>>> import os
```

(continues on next page)

(continued from previous page)

```
>>> key_specs = [{"type": "RSA", "use": ["enc", "sig"]},{"type": "EC", "crv": "P-256",
↪ "use": ["sig"]}]
>>> key_jar = init_key_jar(key_defs=key_specs, private_path='private.jwks', read_
↪only=False)
>>> len(key_jar.get_issuer_keys(''))
3
>>> os.path.isfile('private.jwks')
True
```

To import a JWKS you could do it by first creating a key bundle:

```
>>> from cryptojwt.key_bundle import KeyBundle
>>> from cryptojwt.key_jar import KeyJar
>>> JWKS = {
...     "keys": [
...         {
...          "kty": "RSA",
...          "e": "AQAB",
...          "kid": "abc",
...          "n":
...            "wf-wiusGhA-
↪gleZYQAOPQlNUIucPiqXdPVyieDqQbXXOPBe3nuggtVzeq7pVFH1dZz4dY2Q2LA5DaegvP8kRvoSB_
↪87ds3dy3Rfym_GUSc5B0l1TgEobcyaep8jguRoHto6GWHfCfKqoUYZq4N8vh4LLMQwLR6zi6Jtu82nB5k8"
...     }
...   ]}
>>> kb = KeyBundle(JWKS)
>>> key_jar = KeyJar()
>>> key_jar.add_kb('', kb)
```

Adding a JWKS is such a common thing that there is a simpler way to do it:

```
>>> from cryptojwt.key_bundle import KeyBundle
>>> from cryptojwt.key_issuer import KeyIssuer
>>> from cryptojwt.key_jar import KeyJar
>>> JWKS = {
...     "keys": [
...         {
...          "kty": "RSA",
...          "e": "AQAB",
...          "kid": "abc",
...          "n":
...            "wf-wiusGhA-
↪gleZYQAOPQlNUIucPiqXdPVyieDqQbXXOPBe3nuggtVzeq7pVFH1dZz4dY2Q2LA5DaegvP8kRvoSB_
↪87ds3dy3Rfym_GUSc5B0l1TgEobcyaep8jguRoHto6GWHfCfKqoUYZq4N8vh4LLMQwLR6zi6Jtu82nB5k8"
...     }
...   ]}
>>> kb = KeyBundle(JWKS)
>>> key_jar = KeyJar()
>>> key_jar.import_jwks(JWKS, issuer_id="https://example.com")
>>> sig_keys = key_jar.get_signing_key(key_type="rsa", issuer_id="https://example.com
↪")
>>> len(sig_keys)
1
```

Note Neither variant overwrites anything that was already there.

The end result is the same as when you first created a key bundle and then added it to the key jar.

When dealing with signed and/or encrypted JSON Web Tokens `cryptojwt.key_jar.KeyJar` has these nice methods.

get_jwt_verify_keys `cryptojwt.key_jar.KeyJar.get_jwt_verify_keys()` takes a signed JWT as input and returns a set of keys that can be used to verify the signature. The set you get back is a best estimate and might not contain **the** key. How good the estimate is depends on the information present in the JWS.

get_jwt_decrypt_keys `cryptojwt.key_jar.KeyJar.get_jwt_decrypt_keys()` does the same thing but returns keys that can be used to decrypt a message.

CHAPTER 2

JSON Web Signature (JWS)

JSON Web Signature (JWS) represents content secured with digital signatures or Message Authentication Codes (MACs) using JSON-based data structures.

It is assumed that you know all you need to know about key handling if not please spend some time reading **keyhandling_** .

When it comes to JWS there are basically 2 things you want to be able to do: sign some data and verify that a signature over some data is correct. I'll deal with them in that order.

2.1 Signing a document

There are few steps you have to go through. Let us start with an example and then break it into its parts:

```
>>> from cryptojwt.jwk.hmac import SYMKey
>>> from cryptojwt.jws.jws import JWS

>>> key = SYMKey(key=b'My hollow echo chamber', alg="HS512")
>>> payload = "Please take a moment to register today"
>>> _signer = JWS(payload, alg="HS512")
>>> _jws = _signer.sign_compact([key])
```

The steps:

1. You need keys, one or more. If you provide more than one the software will pick one that has all the necessary qualifications. The keys *MUST* be an instance of `cryptojwt.jwk.JWK` or of a sub class of that class.
2. You need the information that is to be signed. It must be in the form of a string.
3. You initiate the signer, providing it with the message and other needed information.
4. You sign using the compact or the JSON method as described in section 7 of RFC7515 .

2.2 Verifying a signature

Verifying a signature works like this (_jws comes from the first signing example):

```
>>> from cryptojwt.jwk.hmac import SYMKey
>>> from cryptojwt.jws.jws import JWS

>>> key = SYMKey(key=b'My hollow echo chamber', alg="HS512")
>>> _verifier = JWS(alg="HS512")
>>> _msg = _verifier.verify_compact(_jws, [key])
>>> print(_msg)
"Please take a moment to register today"
```

The steps:

1. As with signing, you need a set of keys that can be used to verify the signature. If you provide more than one key, the default is to use them one by one until one works or the list is empty.
2. Initiate the verifier. If you have a reason to expect that a particular signing algorithm is to be used you should give that information to the verifier as shown here. If you don't know, you can leave it out.
3. Verify, using the compact or JSON method.

Or slightly different:

```
>>> from cryptojwt.jws.jws import factory
>>> from cryptojwt.jwk.hmac import SYMKey

>>> key = SYMKey(key=b'My hollow echo chamber', alg="HS512")
>>> _verifier = factory(_jwt)
>>> _verifier.verify_alg('HS512')
True
>>> print(_verifier.verify_compact(_jwt, [key]))
"Please take a moment to register today"
```

Or

```
>>> from cryptojwt.jws.jws import factory
>>> from cryptojwt.jwk.hmac import SYMKey
```

```
>>> key = SYMKey(key=b'My hollow echo chamber', alg="HS512")
>>> _verifier = factory(_jwt, alg="HS512")
>>> print(_verifier.verify_compact(_jwt, [key]))
"Please take a moment to register today"
```

In which case the check of the signing algorithm is done by default.

If you have Key Jar instead of a simple set of keys you can do (not showing how the key jar was initiated here):

```
>>> _verifier = factory(_jws, alg=RS256")
>>> keys = key_jar.get_jwt_verify_keys(_verifier.jwt)
>>> msg = _verifier.verify_compact(token, keys)
```

This is a trick that is used in `cryptojwt.jwt.JWT`

CHAPTER 3

JSON Web Encryption (JWE)

JSON Web Encryption (JWE) represents encrypted content using JSON-based data structures.

It is assumed that you know all you need to know about key handling if not please spend some time reading **keyhandling_** .

When it comes to JWE there are basically 2 things you want to be able to do: encrypt some data and decrypt some encrypted data. I'll deal with them in that order.

3.1 Encrypting a document

This is the high level way of doing things. There are a few steps you have to go through. Let us start with an example and then break it into its parts:

```
>>> from cryptojwt.jwk.rsa import RSAKey
>>> from cryptojwt.jwe.jwe import JWE

>>> priv_key = import_private_rsa_key_from_file(KEY)
>>> pub_key = priv_key.public_key()
>>> encryption_key = RSAKey(use="enc", pub_key=pub_key, kid="some-key-id")
>>> plain = b'Now is the time for all good men to come to the aid of ...'
>>> encryptor = JWE(plain, alg="RSA-OAEP", enc="A256CBC-HS512")
>>> jwe = encryptor.encrypt(keys=[encryption_key], kid="some-key-id")
```

The steps:

1. You need an encryption key. The key *MUST* be an instance of `cryptojwt.jwk.JWK`.
2. You need the information that is to be signed. It must be in the form of a string.
3. You initiate the encryptor, provide it with the message and other needed information.
4. And then you encrypt as described in RFC7516 .

There is a lower level way of doing the same, it will look like this:

```
>>> from cryptojwt.jwk.rsa import import_private_rsa_key_from_file
>>> from cryptojwt.jwe.jwe_rsa import JWE_RSA

>>> priv_key = import_private_rsa_key_from_file('certs/key.pem')
>>> pub_key = priv_key.public_key()
>>> plain = b'Now is the time for all good men to come to the aid of ...'
>>> _rsa = JWE_RSA(plain, alg="RSA-OAEP", enc="A128CBC-HS256")
>>> jwe = _rsa.encrypt(pub_key)
```

Here the key is an cryptography.hazmat.primitives.asymmetric.rsa.RSAPrivateKey instance and the encryptor is a `cryptojwt.jwe.jew_rsa.JWE_RSA` instance.

3.2 Decrypting something encrypted

Decrypting using the encrypted message above.

```
>>> from cryptojwt.jwe.jwe import factory
>>> from cryptojwt.jwk.rsa import RSAKey
```

```
>>> _decryptor = factory(jwe, alg="RSA-OAEP", enc="A128CBC-HS256")
>>> _dkey = RSAKey(priv_key=priv_key)
>>> msg = _decryptor.decrypt(jwe, [_dkey])
```

or if you know what you're doing:

```
>>> _decryptor = JWE_RSA()
>>> msg = _decryptor.decrypt(jwe, priv_key)
```

CHAPTER 4

CryptoJWT classes and functions documentation!

Description of CryptoJWT classes and functions.

Contents:

4.1 cryptojwt.key_bundle package

4.1.1 Module contents

4.2 cryptojwt.key_jar package

4.2.1 Module contents

4.3 cryptojwt.jwk package

4.3.1 Submodules

4.3.2 cryptojwt.jwk.asym module

4.3.3 cryptojwt.jwk.ec module

4.3.4 cryptojwt.jwk.hmac module

4.3.5 cryptojwt.jwk.jwk module

4.3.6 cryptojwt.jwk.rsa module

4.3.7 cryptojwt.jwk.utils module

4.4 cryptojwt.simple_jwt package

4.4.1 Module contents

4.5 cryptojwt.jwx package

4.5.1 Module contents

4.6 cryptojwt.jws package

4.6.1 Submodules

4.6.2 cryptojwt.jws.dsa module

4.6.3 cryptojwt.jws.exception module

4.6.4 cryptojwt.jws.hmac module

4.6.5 cryptojwt.jws.jws module

4.6.6 cryptojwt.jws.pss module

4.6.7 cryptojwt.jws.rsa module

4.6.8 cryptojwt.jws.utils module

CHAPTER 5

Indices and tables

- genindex
- modindex
- search