PyJWT Documentation

Release 1.5.3

José Padilla

Sep 05, 2017

Contents

PyJWT is a Python library which allows you to encode and decode JSON Web Tokens (JWT). JWT is an open, industry-standard (RFC 7519) for representing claims securely between two parties.

CHAPTER 1

Installation

You can install `pyjwt` with `pip`:

```
$ pip install pyjwt
```

See *Installation* for more information.

CHAPTER 2

Example Usage

```
>>> import jwt

>>> encoded_jwt = jwt.encode({'some': 'payload'}, 'secret', algorithm='HS256')
>>> encoded_jwt
'eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzb21lIjoicGF5bG9hZCJ9.
↪4twFt5NiznN84AWoo1d7KO1T_yoc0Z6XOpOVswacPZg'

>>> jwt.decode(encoded_jwt, 'secret', algorithms=['HS256'])
{'some': 'payload'}
```

See *Usage Examples* for more examples.

CHAPTER 3

Command line

Usage:

```
pyjwt [options] INPUT
```

Decoding examples:

```
pyjwt --key=secret TOKEN
pyjwt --no-verify TOKEN
```

See more options executing `pyjwt --help`.

CHAPTER 4

Index

Installation

You can install `PyJWT` with `pip`:

```
$ pip install pyjwt
```

Cryptographic Dependencies (Optional)

If you are planning on encoding or decoding tokens using certain digital signature algorithms (like RSA or ECDSA), you will need to install the cryptography library.

```
$ pip install cryptography
```

Legacy Dependencies

Some environments, most notably Google App Engine, do not allow the installation of Python packages that require compilation of C extensions and therefore cannot install `cryptography`. If you can install `cryptography`, you should disregard this section.

If you are deploying an application to one of these environments, you may need to use the legacy implementations of the digital signature algorithms:

```
$ pip install pycrypto ecdsa
```

Once you have installed `pycrypto` and `ecdcsa`, you can tell PyJWT to use the legacy implementations with `jwt.register_algorithm()`. The following example code shows how to configure PyJWT to use the legacy implementations for RSA with SHA256 and EC with SHA256 signatures.

```
import jwt
from jwt.contrib.algorithms.pycrypto import RSAAlgorithm
from jwt.contrib.algorithms.py_ecdsa import ECAlgorithm

jwt.register_algorithm('RS256', RSAAlgorithm(RSAAlgorithm.SHA256))
jwt.register_algorithm('ES256', ECAlgorithm(ECAlgorithm.SHA256))
```

Usage Examples

Encoding & Decoding Tokens

```
>>import jwt
>>encoded = jwt.encode({'some': 'payload'}, 'secret', algorithm='HS256')
'eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzb21lIjoicGF5bG9hZCJ9.
↪4twFt5NiznN84AWoo1d7KO1T_yoc0Z6XOpOVswacPZg'
```

Specifying Additional Headers

```
>>jwt.encode({'some': 'payload'}, 'secret', algorithm='HS256', headers={'kid':
↪'230498151c214b788dd97f22b85410a5'})

↪'eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCIsImtpZCI6IjIzMDQ5ODE1MWMyMTRiNzg4ZGQ5N2YyMmI4NTQxMGE1In0.
↪eyJzb21lIjoicGF5bG9hZCJ9.DogbDGmMHgA_bU05TAB-R6geQ2nMU2BRM-LnYEtefwg'
```

Reading the Claimset without Validation

If you wish to read the claimset of a JWT without performing validation of the signature or any of the registered claim names, you can set the `verify` parameter to `False`.

Note: It is generally ill-advised to use this functionality unless you clearly understand what you are doing. Without digital signature information, the integrity or authenticity of the claimset cannot be trusted.

```
>>jwt.decode(encoded, verify=False)
{u'some': u'payload'}
```

Registered Claim Names

The JWT specificaftion defines some registered claim names and defines how they should be used. PyJWT supports these registered claim names:

- “exp” (Expiration Time) Claim
- “nbf” (Not Before Time) Claim
- “iss” (Issuer) Claim
- “aud” (Audience) Claim
- “iat” (Issued At) Claim

Expiration Time Claim (exp)

> The "exp" (expiration time) claim identifies the expiration time on or after which the JWT MUST NOT be accepted for processing. The processing of the "exp" claim requires that the current date/time MUST be before the expiration date/time listed in the "exp" claim. Implementers MAY provide for some small leeway, usually no more than a few minutes, to account for clock skew. Its value MUST be a number containing a NumericDate value. Use of this claim is OPTIONAL.

You can pass the expiration time as a UTC UNIX timestamp (an int) or as a datetime, which will be converted into an int. For example:

```
jwt.encode({'exp': 1371720939}, 'secret')
jwt.encode({'exp': datetime.utcnow()}, 'secret')
```

Expiration time is automatically verified in *jwt.decode()* and raises *jwt.ExpiredSignatureError* if the expiration time is in the past:

```
try:
    jwt.decode('JWT_STRING', 'secret', algorithms=['HS256'])
except jwt.ExpiredSignatureError:
    # Signature has expired
```

Expiration time will be compared to the current UTC time (as given by *timegm(datetime.utcnow().utctimetuple())*), so be sure to use a UTC timestamp or datetime in encoding.

You can turn off expiration time verification with the *verify_exp* parameter in the options argument.

PyJWT also supports the leeway part of the expiration time definition, which means you can validate a expiration time which is in the past but not very far. For example, if you have a JWT payload with a expiration time set to 30 seconds after creation but you know that sometimes you will process it after 30 seconds, you can set a leeway of 10 seconds in order to have some margin:

```
jwt_payload = jwt.encode({
    'exp': datetime.datetime.utcnow() + datetime.timedelta(seconds=30)
}, 'secret')

time.sleep(32)

# JWT payload is now expired
# But with some leeway, it will still validate
jwt.decode(jwt_payload, 'secret', leeway=10, algorithms=['HS256'])
```

Instead of specifying the leeway as a number of seconds, a *datetime.timedelta* instance can be used. The last line in the example above is equivalent to:

```
jwt.decode(jwt_payload, 'secret', leeway=datetime.timedelta(seconds=10), algorithms=[
↪'HS256'])
```

Not Before Time Claim (nbf)

> The "nbf" (not before) claim identifies the time before which the JWT MUST NOT be accepted for processing. The processing of the "nbf" claim requires that the current date/time MUST be after or equal to the not-before date/time listed in the "nbf" claim. Implementers MAY provide for some small leeway, usually no more than a few minutes, to account for clock skew. Its value MUST be a number containing a NumericDate value. Use of this claim is OPTIONAL.

The *nbf* claim works similarly to the *exp* claim above.

```
jwt.encode({'nbf': 1371720939}, 'secret')
jwt.encode({'nbf': datetime.utcnow()}, 'secret')
```

Issuer Claim (iss)

The "iss" (issuer) claim identifies the principal that issued the JWT. The processing of this claim is generally application specific. The "iss" value is a case-sensitive string containing a StringOrURI value. Use of this claim is OPTIONAL.

```
payload = {
    'some': 'payload',
    'iss': 'urn:foo'
}

token = jwt.encode(payload, 'secret')
decoded = jwt.decode(token, 'secret', issuer='urn:foo', algorithms=['HS256'])
```

If the issuer claim is incorrect, *jwt.InvalidIssuerError* will be raised.

Audience Claim (aud)

The "aud" (audience) claim identifies the recipients that the JWT is intended for. Each principal intended to process the JWT MUST identify itself with a value in the audience claim. If the principal processing the claim does not identify itself with a value in the "aud" claim when this claim is present, then the JWT MUST be rejected. In the general case, the "aud" value is an array of case- sensitive strings, each containing a StringOrURI value. In the special case when the JWT has one audience, the "aud" value MAY be a single case-sensitive string containing a StringOrURI value. The interpretation of audience values is generally application specific. Use of this claim is OPTIONAL.

```
payload = {
    'some': 'payload',
    'aud': 'urn:foo'
}

token = jwt.encode(payload, 'secret')
decoded = jwt.decode(token, 'secret', audience='urn:foo', algorithms=['HS256'])
```

If the audience claim is incorrect, *jwt.InvalidAudienceError* will be raised.

Issued At Claim (iat)

The iat (issued at) claim identifies the time at which the JWT was issued. This claim can be used to determine the age of the JWT. Its value MUST be a number containing a NumericDate value. Use of this claim is OPTIONAL.

If the *iat* claim is not a number, an *jwt.InvalidIssuedAtError* exception will be raised.

```
jwt.encode({'iat': 1371720939}, 'secret')
jwt.encode({'iat': datetime.utcnow()}, 'secret')
```

Frequently Asked Questions

How can I extract a public / private key from a x509 certificate?

The `load_pem_x509_certificate()` function from `cryptography` can be used to extract the public or private keys from a x509 certificate in PEM format.

```
# Python 2
from cryptography.x509 import load_pem_x509_certificate
from cryptography.hazmat.backends import default_backend

cert_str = "-----BEGIN CERTIFICATE-----MIIDETCCAfm..."
cert_obj = load_pem_x509_certificate(cert_str, default_backend())
public_key = cert_obj.public_key()
private_key = cert_obj.private_key()
```

```
# Python 3
from cryptography.x509 import load_pem_x509_certificate
from cryptography.hazmat.backends import default_backend

cert_str = "-----BEGIN CERTIFICATE-----MIIDETCCAfm...".encode()
cert_obj = load_pem_x509_certificate(cert_str, default_backend())
public_key = cert_obj.public_key()
private_key = cert_obj.private_key()
```

I'm using Google App Engine and can't install *cryptography*, what can I do?

Some platforms like Google App Engine don't allow you to install libraries that require C extensions to be built (like *cryptography*). If you're deploying to one of those environments, you should check out *Legacy Dependencies*

Digital Signature Algorithms

The JWT specification supports several algorithms for cryptographic signing. This library currently supports:

- HS256 - HMAC using SHA-256 hash algorithm (default)
- HS384 - HMAC using SHA-384 hash algorithm
- HS512 - HMAC using SHA-512 hash algorithm
- ES256 - ECDSA signature algorithm using SHA-256 hash algorithm
- ES384 - ECDSA signature algorithm using SHA-384 hash algorithm
- ES512 - ECDSA signature algorithm using SHA-512 hash algorithm
- RS256 - RSASSA-PKCS1-v1_5 signature algorithm using SHA-256 hash algorithm
- RS384 - RSASSA-PKCS1-v1_5 signature algorithm using SHA-384 hash algorithm
- RS512 - RSASSA-PKCS1-v1_5 signature algorithm using SHA-512 hash algorithm
- PS256 - RSASSA-PSS signature using SHA-256 and MGF1 padding with SHA-256
- PS384 - RSASSA-PSS signature using SHA-384 and MGF1 padding with SHA-384
- PS512 - RSASSA-PSS signature using SHA-512 and MGF1 padding with SHA-512

Asymmetric (Public-key) Algorithms

Usage of RSA (RS*) and EC (EC*) algorithms require a basic understanding of how public-key cryptography is used with regards to digital signatures. If you are unfamiliar, you may want to read [this article](http://en.wikipedia.org/wiki/Public-key_cryptography).

When using the RSASSA-PKCS1-v1_5 algorithms, the *key* argument in both `jwt.encode()` and `jwt.decode()` (`"secret"` in the examples) is expected to be either an RSA public or private key in PEM or SSH format. The type of key (private or public) depends on whether you are signing or verifying a token.

When using the ECDSA algorithms, the `key` argument is expected to be an Elliptic Curve public or private key in PEM format. The type of key (private or public) depends on whether you are signing or verifying.

Specifying an Algorithm

You can specify which algorithm you would like to use to sign the JWT by using the *algorithm* parameter:

```
>>> encoded = jwt.encode({'some': 'payload'}, 'secret', algorithm='HS512')
'eyJhbGciOiJIUzUxMiIsInR5cCI6IkpXVCJ9.eyJzb21lIjoicGF5bG9hZCJ9.
↪WTzLzFO079PduJiFIyzrOah54YaM8qoxH9fLMQoQhKtw3_
↪fMGjImIOokijDkXVbyfBqhMo2GCNu4w9v7UXvnpA'
```

When decoding, you can also specify which algorithms you would like to permit when validating the JWT by using the *algorithms* parameter which takes a list of allowed algorithms:

```
>>> jwt.decode(encoded, 'secret', algorithms=['HS512', 'HS256'])
{u'some': u'payload'}
```

In the above case, if the JWT has any value for its alg header other than HS512 or HS256, the claim will be rejected with an `InvalidAlgorithmError`.

API Reference

TODO: Document PyJWS / PyJWT classes

Exceptions

class `jwt.exceptions.`**`InvalidTokenError`**
Base exception when `decode()` fails on a token

class `jwt.exceptions.`**`DecodeError`**
Raised when a token cannot be decoded because it failed validation

class `jwt.exceptions.`**`ExpiredSignatureError`**
Raised when a token's `exp` claim indicates that it has expired

class `jwt.exceptions.`**`InvalidAudienceError`**
Raised when a token's `aud` claim does not match one of the expected audience values

class `jwt.exceptions.`**`InvalidIssuerError`**
Raised when a token's `iss` claim does not match the expected issuer

class `jwt.exceptions.`**`InvalidIssuedAtError`**
Raised when a token's `iat` claim is in the future

class jwt.exceptions.**ImmatureSignatureError**
Raised when a token's nbf claim represents a time in the future

class jwt.exceptions.**InvalidKeyError**
Raised when the specified key is not in the proper format

class jwt.exceptions.**InvalidAlgorithmError**
Raised when the specified algorithm is not recognized by PyJWT

class jwt.exceptions.**MissingRequiredClaimError**
Raised when a claim that is required to be present is not contained in the claimset

Python Module Index

j

Index