PyJWT

Release 2.2.0

Oct 06, 2021

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

Sponsor

If you want to quickly add secure token-based authentication to Python projects, feel free to check Auth0's Python SDK and free plan at auth0.com/developers.

CHAPTER 2

Installation

You can install `pyjwt` with `pip`:

```
$ pip install pyjwt
```

See *Installation* for more information.

CHAPTER 3

Example Usage

```
>>> import jwt
>>> encoded_jwt = jwt.encode({"some": "payload"}, "secret", algorithm="HS256")
>>> print(encoded_jwt)
eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJzb21lIjoicGF5bG9hZCJ9.
↪Joh1R2dYzkRvDkqv3sygm5YyK8Gi4ShZqbhK2gxcs2U
>>> jwt.decode(encoded_jwt, "secret", algorithms=["HS256"])
{'some': 'payload'}
```

See *Usage Examples* for more examples.

CHAPTER 4

Index

4.1 Installation

You can install `PyJWT` with `pip`:

```
$ pip install pyjwt
```

4.1.1 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. This can be installed explicitly, or as a required extra in the `pyjwt` requirement:

```
$ pip install pyjwt[crypto]
```

The `pyjwt[crypto]` format is recommended in requirements files in projects using `PyJWT`, as a separate `cryptography` requirement line may later be mistaken for an unused requirement and removed.

4.2 Usage Examples

4.2.1 Encoding & Decoding Tokens with HS256

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

4.2.2 Encoding & Decoding Tokens with RS256 (RSA)

RSA encoding and decoding require the `cryptography` module. See *Cryptographic Dependencies (Optional)*.

```
>>> import jwt
>>> private_key = b"-----BEGIN PRIVATE KEY-----\nMIGEAgEAMBAGByqGSM49AgEGBS..."
>>> public_key = b"-----BEGIN PUBLIC KEY-----\nMHYwEAYHKoZIzj0CAQYFK4EEAC..."
>>> encoded = jwt.encode({"some": "payload"}, private_key, algorithm="RS256")
>>> print(encoded)
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzb21lIjoicGF5bG9hZCJ9.
↪4twFt5NiznN84AWoo1d7KO1T_yoc0Z6XOpOVswacPZg
>>> decoded = jwt.decode(encoded, public_key, algorithms=["RS256"])
{'some': 'payload'}
```

If your private key needs a passphrase, you need to pass in a `PrivateKey` object from `cryptography`.

```
from cryptography.hazmat.primitives import serialization
from cryptography.hazmat.backends import default_backend

pem_bytes = b"-----BEGIN PRIVATE KEY-----\nMIGEAgEAMBAGByqGSM49AgEGBS..."
passphrase = b"your password"

private_key = serialization.load_pem_private_key(
    pem_bytes, password=passphrase, backend=default_backend()
)
encoded = jwt.encode({"some": "payload"}, private_key, algorithm="RS256")
```

4.2.3 Specifying Additional Headers

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

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

4.2.4 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_signature` option 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, options={"verify_signature": False})
{'some': 'payload'}
```

4.2.5 Reading Headers without Validation

Some APIs require you to read a JWT header without validation. For example, in situations where the token issuer uses multiple keys and you have no way of knowing in advance which one of the issuer's public keys or shared secrets

to use for validation, the issuer may include an identifier for the key in the header.

```
>>> jwt.get_unverified_header(encoded)
{'alg': 'RS256', 'typ': 'JWT', 'kid': 'key-id-12345...'}
```

4.2.6 Registered Claim Names

The JWT specification 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.now(tz=timezone.utc)}, "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.now(tz=timezone.utc).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.now(tz=timezone.utc) + datetime.timedelta(seconds=30)},
    "secret",
```

(continues on next page)

(continued from previous page)

```
)

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.now(tz=timezone.utc)}, "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.

```
payload = {"some": "payload", "aud": ["urn:foo", "urn:bar"]}

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

In the special case when the JWT has one audience, the "aud" value MAY be a single case-sensitive string containing a StringOrURI value.

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

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

If multiple audiences are accepted, the `audience` parameter for `jwt.decode` can also be an iterable

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

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

The interpretation of audience values is generally application specific. Use of this claim is OPTIONAL.

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.now(tz=timezone.utc)}, "secret")
```

4.2.7 Requiring Presence of Claims

If you wish to require one or more claims to be present in the claimset, you can set the `require` parameter to include these claims.

```
>>> jwt.decode(encoded, options={"require": ["exp", "iss", "sub"]})
{'exp': 1371720939, 'iss': 'urn:foo', 'sub': '25c37522-f148-4cbf-8ee6-c4a9718dd0af'}
```

4.2.8 Retrieve RSA signing keys from a JWKS endpoint

```
>>> import jwt
>>> from jwt import PyJWKClient
>>> token =
↪"eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiIsImtpZCI6Ik5FRTFRVVJCT1RNNE16STVSa0ZETlRZeE9UVTFNRGcyT0Rnd1EwV
↪eyJpc3MiOiJodHRwczovL2Rldi04N2V2eDlydS5hdXRoMC5jb20vIiwic3ViIjoiYVc0Q2NhNzl4UmVMV1V6MGFFMkg2a0QwTzN
↪PUxE7xn52aTCohGiWoSdMBZGiYAHwE5FYie0Y1qUT68IHSTXwXVd6hn02HTah6epvHHVKA2FqcFZ4GGv5VTHEvYpeggiiZMgbxH
↪GXvgFzsdsyWlVQvPX2xjeaQ217r2PtxDeqjlf66UYl6oY6AqNS8DH3iryCvIfCcybRZkc_hdv...
↪6ZMoKT6Piijvk_aXdm7-QQqKJFHLuEqrVSOuBqqiNfVrG27QzAPuPOxvfXTVLXL2jek5meH6n-
↪VWgrBdoMFH93QEszEDowDAEhQPHVs0xj7SIzA"
```

(continues on next page)

(continued from previous page)

```
>>> kid = "NEE1QURBOTM4MzI5RkFDNTYxOTU1MDg2ODgwQ0UzMTk1QjYyRkRFQw"
>>> url = "https://dev-87evx9ru.auth0.com/.well-known/jwks.json"
>>> jwks_client = PyJWKClient(url)
>>> signing_key = jwks_client.get_signing_key_from_jwt(token)
>>> data = jwt.decode(
...     token,
...     signing_key.key,
...     algorithms=["RS256"],
...     audience="https://expenses-api",
...     options={"verify_exp": False},
... )
>>> print(data)
{'iss': 'https://dev-87evx9ru.auth0.com/', 'sub':
↪'aW4Cca79xReLWUz0aE2H6kD0O3cXBVtC@clients', 'aud': 'https://expenses-api', 'iat':␣
↪1572006954, 'exp': 1572006964, 'azp': 'aW4Cca79xReLWUz0aE2H6kD0O3cXBVtC', 'gty':
↪'client-credentials'}
```

4.3 Frequently Asked Questions

4.3.1 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.

```
from cryptography.x509 import load_pem_x509_certificate

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

4.4 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
- ES256K - ECDSA signature algorithm with secp256k1 curve 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
- EdDSA - Both Ed25519 signature using SHA-512 and Ed448 signature using SHA-3 are supported. Ed25519 and Ed448 provide 128-bit and 224-bit security respectively.

4.4.1 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.

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.

4.4.2 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")
>>> print(encoded)
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"])
{'some': '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`.

Warning: Do **not** compute the `algorithms` parameter based on the `alg` from the token itself, or on any other data that an attacker may be able to influence, as that might expose you to various vulnerabilities (see RFC 8725 §2.1). Instead, either hard-code a fixed value for `algorithms`, or configure it in the same place you configure the `key`. Make sure not to mix symmetric and asymmetric algorithms that interpret the `key` in different ways (e.g. HS* and RS*).

4.5 API Reference

`jwt.`**`encode`**(*payload*, *key*, *algorithm="HS256"*, *headers=None*, *json_encoder=None*)

Encode the `payload` as JSON Web Token.

Parameters

- **`payload`** (*dict*) – JWT claims, e.g. `dict(iss=..., aud=..., sub=...)`
- **`key`** (*str*) – a key suitable for the chosen algorithm:

- for **asymmetric algorithms**: PEM-formatted private key, a multiline string
- for **symmetric algorithms**: plain string, sufficiently long for security

- **algorithm** (*str*) – algorithm to sign the token with, e.g. `"ES256"`. If `headers` includes `alg`, it will be preferred to this parameter.
- **headers** (*dict*) – additional JWT header fields, e.g. `dict(kid="my-key-id")`.
- **json_encoder** (*json.JSONEncoder*) – custom JSON encoder for `payload` and `headers`

Return type str

Returns a JSON Web Token

`jwt.`**decode**(*jwt*, *key=""*, *algorithms=None*, *options=None*, *audience=None*, *issuer=None*, *leeway=0*)

Verify the `jwt` token signature and return the token claims.

Parameters

- **jwt** (*str*) – the token to be decoded
- **key** (*str*) – the key suitable for the allowed algorithm
- **algorithms** (*list*) – allowed algorithms, e.g. `["ES256"]`

> **Warning:** Do **not** compute the `algorithms` parameter based on the `alg` from the token itself, or on any other data that an attacker may be able to influence, as that might expose you to various vulnerabilities (see RFC 8725 §2.1). Instead, either hard-code a fixed value for `algorithms`, or configure it in the same place you configure the `key`. Make sure not to mix symmetric and asymmetric algorithms that interpret the `key` in different ways (e.g. HS* and RS*).

- **options** (*dict*) – extended decoding and validation options
  - `verify_signature=True` verify the JWT cryptographic signature
  - `require=[]` list of claims that must be present. Example: `require=["exp", "iat", "nbf"]`. **Only verifies that the claims exists**. Does not verify that the claims are valid.
  - `verify_aud=verify_signature` check that `aud` (audience) claim matches `audience`
  - `verify_iss=verify_signature` check that `iss` (issuer) claim matches `issuer`
  - `verify_exp=verify_signature` check that `exp` (expiration) claim value is in the future
  - `verify_iat=verify_signature` check that `iat` (issued at) claim value is an integer
  - `verify_nbf=verify_signature` check that `nbf` (not before) claim value is in the past

> **Warning:** `exp`, `iat` and `nbf` will only be verified if present. Please pass respective value to `require` if you want to make sure that they are always present (and therefore always verified if `verify_exp`, `verify_iat`, and `verify_nbf` respectively is set to `True`).

- **audience** (*Iterable*) – optional, the value for `verify_aud` check
- **issuer** (*str*) – optional, the value for `verify_iss` check
- **leeway** (*float*) – a time margin in seconds for the expiration check

Return type dict

Returns the JWT claims

`jwt.`**decode_complete**(*jwt*, *key=""*, *algorithms=None*, *options=None*, *audience=None*, *issuer=None*, *leeway=0*)

Identical to `jwt.decode` except for return value which is a dictionary containing the token header (JOSE Header), the token payload (JWT Payload), and token signature (JWT Signature) on the keys "header", "payload", and "signature" respectively.

Parameters

- **jwt** (*str*) – the token to be decoded
- **key** (*str*) – the key suitable for the allowed algorithm
- **algorithms** (*list*) – allowed algorithms, e.g. `["ES256"]`

> **Warning:** Do **not** compute the `algorithms` parameter based on the `alg` from the token itself, or on any other data that an attacker may be able to influence, as that might expose you to various vulnerabilities (see RFC 8725 §2.1). Instead, either hard-code a fixed value for `algorithms`, or configure it in the same place you configure the `key`. Make sure not to mix symmetric and asymmetric algorithms that interpret the `key` in different ways (e.g. HS* and RS*).

- **options** (*dict*) – extended decoding and validation options
  - `verify_signature=True` verify the JWT cryptographic signature
  - `require=[]` list of claims that must be present. Example: `require=["exp", "iat", "nbf"]`. **Only verifies that the claims exists**. Does not verify that the claims are valid.
  - `verify_aud=verify_signature` check that `aud` (audience) claim matches `audience`
  - `verify_iss=verify_signature` check that `iss` (issuer) claim matches `issuer`
  - `verify_exp=verify_signature` check that `exp` (expiration) claim value is in the future
  - `verify_iat=verify_signature` check that `iat` (issued at) claim value is an integer
  - `verify_nbf=verify_signature` check that `nbf` (not before) claim value is in the past

> **Warning:** `exp`, `iat` and `nbf` will only be verified if present. Please pass respective value to `require` if you want to make sure that they are always present (and therefore always verified if `verify_exp`, `verify_iat`, and `verify_nbf` respectively is set to `True`).

- **audience** (*Iterable*) – optional, the value for `verify_aud` check

- **issuer** (*str*) – optional, the value for `verify_iss` check
- **leeway** (*float*) – a time margin in seconds for the expiration check

Return type dict

Returns Decoded JWT with the JOSE Header on the key `header`, the JWS Payload on the key `payload`, and the JWS Signature on the key `signature`.

Note: TODO: Document PyJWS class

4.5.1 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.`**InvalidSignatureError**
Raised when a token's signature doesn't match the one provided as part of the token.

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

4.6 Changelog

All notable changes to this project will be documented in this file. This project adheres to Semantic Versioning.

4.6.1 Unreleased

Changed

Fixed

Added

4.6.2 v2.2.0

Changed

- Remove arbitrary kwargs. #657
- Use timezone package as Python 3.5+ is required. #694

Fixed

- Assume JWK without the "use" claim is valid for signing as per RFC7517 #668
- Prefer *headers["alg"]* to *algorithm* in *jwt.encode()*. #673
- Fix aud validation to support {'aud': null} case. #670
- Make *typ* optional in JWT to be compliant with RFC7519. #644
- Remove upper bound on cryptography version. #693

Added

- Add support for Ed448/EdDSA. #675

4.6.3 v2.1.0

Changed

- Allow claims validation without making JWT signature validation mandatory. #608

Fixed

- Remove padding from JWK test data. #628
- Make *kty* mandatory in JWK to be compliant with RFC7517. #624
- Allow JWK without *alg* to be compliant with RFC7517. #624
- Allow to verify with private key on ECAlgorithm, as well as on Ed25519Algorithm. #645

Added

- Add caching by default to PyJWKClient #611
- Add missing exceptions.InvalidKeyError to jwt module __init__ imports #620
- Add support for ES256K algorithm #629
- Add *from_jwk()* to Ed25519Algorithm #621
- Add *to_jwk()* to Ed25519Algorithm #643
- Export *PyJWK* and *PyJWKSet* #652

4.6.4 v2.0.1

Changed

- Rename CHANGELOG.md to CHANGELOG.rst and include in docs #597

Fixed

- Fix *from_jwk()* for all algorithms #598

Added

4.6.5 v2.0.0

Changed

Drop support for Python 2 and Python 3.0-3.5

Python 3.5 is EOL so we decide to drop its support. Version `1.7.1` is the last one supporting Python 3.0-3.5.

Require cryptography >= 3

Drop support for PyCrypto and ECDSA

We've kept this around for a long time, mostly for environments that didn't allow installing cryptography.

Drop CLI

Dropped the included cli entry point.

Improve typings

We no longer need to use mypy Python 2 compatibility mode (comments)

`jwt.encode(...)` return type

Tokens are returned as string instead of a byte string

Dropped deprecated errors

Removed `ExpiredSignature`, `InvalidAudience`, and `InvalidIssuer`. Use `ExpiredSignatureError`, `InvalidAudienceError`, and `InvalidIssuerError` instead.

Dropped deprecated `verify_expiration` param in `jwt.decode(...)`

Use `jwt.decode(encoded, key, algorithms=["HS256"], options={"verify_exp": False})` instead.

Dropped deprecated `verify` param in `jwt.decode(...)`

Use `jwt.decode(encoded, key, options={"verify_signature": False})` instead.

Require explicit `algorithms` in `jwt.decode(...)` by default

Example: `jwt.decode(encoded, key, algorithms=["HS256"])`.

Dropped deprecated `require_*` options in `jwt.decode(...)`

For example, instead of `jwt.decode(encoded, key, algorithms=["HS256"], options={"require_exp": True})`, use `jwt.decode(encoded, key, algorithms=["HS256"], options={"require": ["exp"]})`.

Added

Introduce better experience for JWKs

Introduce `PyJWK`, `PyJWKSet`, and `PyJWKClient`.

```
import jwt
from jwt import PyJWKClient

token =
↪"eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiIsImtpZCI6Ik5FRTFRVVJCT1RNNE16STVSa0ZETlRZeE9UVTFNRGcyT0Rnd1EwV
↪eyJpc3MiOiJodHRwczovL2Rldi04N2V2eDlydS5hdXRoMC5jb20vIiwic3ViIjoiYVc0Q2NhNzl4UmVMV1V6MGFFMkg2a0QwTz
↪PUxE7xn52aTCohGiWoSdMBZGiYAHwE5FYie0Y1qUT68IHSTXwXVd6hn02HTah6epvHHVKA2FqcFZ4GGv5VTHEvYpeggiiZMgbx
↪GXvgFzsdsyWlVQvPX2xjeaQ217r2PtxDeqjlf66UYl6oY6AqNS8DH3iryCvIfCcybRZkc_hdy-
↪6ZMoKT6Piijvk_aXdm7-QQqKJFHLuEqrVSOuBqqiNfVrG27QzAPuPOxvfXTVLXL2jek5meH6n-
↪VWgrBdoMFH93QEszEDowDAEhQPHVs0xj7SIzA"
kid = "NEE1QURBOTM4MzI5RkFDNTYxOTU1MDg2ODgwQ0UzMTk1QjYyRkRFQw"
url = "https://dev-87evx9ru.auth0.com/.well-known/jwks.json"

jwks_client = PyJWKClient(url)
signing_key = jwks_client.get_signing_key_from_jwt(token)

data = jwt.decode(
    token,
    signing_key.key,
    algorithms=["RS256"],
    audience="https://expenses-api",
    options={"verify_exp": False},
)
print(data)
```

Support for JWKs containing ECDSA keys

Add support for Ed25519 / EdDSA

Pull Requests

- Add PyPy3 to the test matrix (#550) by @jdufresne
- Require tweak (#280) by @psafont
- Decode return type is dict[str, Any] (#393) by @jacopofar
- Fix linter error in test_cli (#414) by @jaraco
- Run mypy with tox (#421) by @jpadilla
- Document (and prefer) pyjwt[crypto] req format (#426) by @gthb
- Correct type for json_encoder argument (#438) by @jdufresne
- Prefer https:// links where available (#439) by @jdufresne
- Pass python_requires argument to setuptools (#440) by @jdufresne
- Rename [wheel] section to [bdist_wheel] as the former is legacy (#441) by @jdufresne
- Remove setup.py test command in favor of pytest and tox (#442) by @jdufresne
- Fix mypy errors (#449) by @jpadilla
- DX Tweaks (#450) by @jpadilla
- Add support of python 3.8 (#452) by @Djailla
- Fix 406 (#454) by @justinbaur
- Add support for Ed25519 / EdDSA, with unit tests (#455) by @Someguy123
- Remove Python 2.7 compatibility (#457) by @Djailla
- Fix simple typo: encododed -> encoded (#462) by @timgates42
- Enhance tracebacks. (#477) by @JulienPalard
- Simplify `python_requires` (#478) by @michael-k
- Document top-level .encode and .decode to close #459 (#482) by @dimaqq
- Improve documentation for audience usage (#484) by @CorreyL
- Correct README on how to run tests locally (#489) by @jdufresne
- Fix `tox -e lint` warnings and errors (#490) by @jdufresne
- Run pyupgrade across project to use modern Python 3 conventions (#491) by @jdufresne
- Add Python-3-only trove classifier and remove “universal” from wheel (#492) by @jdufresne
- Emit warnings about user code, not pyjwt code (#494) by @mgedmin
- Move setup information to declarative setup.cfg (#495) by @jdufresne
- CLI options for verifying audience and issuer (#496) by @GeoffRichards
- Specify the target Python version for mypy (#497) by @jdufresne
- Remove unnecessary compatibility shims for Python 2 (#498) by @jdufresne
- Setup GH Actions (#499) by @jpadilla

- Implementation of ECAlgorithm.from_jwk (#500) by @jpadilla
- Remove cli entry point (#501) by @jpadilla
- Expose InvalidKeyError on jwt module (#503) by @russellcardullo
- Avoid loading token twice in pyjwt.decode (#506) by @CaselIT
- Default links to stable version of documentation (#508) by @salcedo
- Update README.md badges (#510) by @jpadilla
- Introduce better experience for JWKs (#511) by @jpadilla
- Fix tox conditional extras (#512) by @jpadilla
- Return tokens as string not bytes (#513) by @jpadilla
- Drop support for legacy contrib algorithms (#514) by @jpadilla
- Drop deprecation warnings (#515) by @jpadilla
- Update Auth0 sponsorship link (#519) by @Sambego
- Update return type for jwt.encode (#521) by @moomoolive
- Run tests against Python 3.9 and add trove classifier (#522) by @michael-k
- Removed redundant `default_backend()` (#523) by @rohitkg98
- Documents how to use private keys with passphrases (#525) by @rayluo
- Update version to 2.0.0a1 (#528) by @jpadilla
- Fix usage example (#530) by @nijel
- add EdDSA to docs (#531) by @CircleOnCircles
- Remove support for EOL Python 3.5 (#532) by @jdufresne
- Upgrade to isort 5 and adjust configurations (#533) by @jdufresne
- Remove unused argument “verify” from PyJWS.decode() (#534) by @jdufresne
- Update typing syntax and usage for Python 3.6+ (#535) by @jdufresne
- Run pyupgrade to simplify code and use Python 3.6 syntax (#536) by @jdufresne
- Drop unknown pytest config option: strict (#537) by @jdufresne
- Upgrade black version and usage (#538) by @jdufresne
- Remove “Command line” sections from docs (#539) by @jdufresne
- Use existing key_path() utility function throughout tests (#540) by @jdufresne
- Replace force_bytes()/force_unicode() in tests with literals (#541) by @jdufresne
- Remove unnecessary Unicode decoding before json.loads() (#542) by @jdufresne
- Remove unnecessary force_bytes() calls priot to base64url_decode() (#543) by @jdufresne
- Remove deprecated arguments from docs (#544) by @jdufresne
- Update code blocks in docs (#545) by @jdufresne
- Refactor jwt/jwks_client.py without requests dependency (#546) by @jdufresne
- Tighten bytes/str boundaries and remove unnecessary coercing (#547) by @jdufresne
- Replace codecs.open() with builtin open() (#548) by @jdufresne

- Replace int_from_bytes() with builtin int.from_bytes() (#549) by @jdufresne
- Enforce .encode() return type using mypy (#551) by @jdufresne
- Prefer direct indexing over options.get() (#552) by @jdufresne
- Cleanup "noqa" comments (#553) by @jdufresne
- Replace merge_dict() with builtin dict unpacking generalizations (#555) by @jdufresne
- Do not mutate the input payload in PyJWT.encode() (#557) by @jdufresne
- Use direct indexing in PyJWKClient.get_signing_key_from_jwt() (#558) by @jdufresne
- Split PyJWT/PyJWS classes to tighten type interfaces (#559) by @jdufresne
- Simplify mocked_response test utility function (#560) by @jdufresne
- Autoupdate pre-commit hooks and apply them (#561) by @jdufresne
- Remove unused argument "payload" from PyJWS.*verify*signature() (#562) by @jdufresne
- Add utility functions to assist test skipping (#563) by @jdufresne
- Type hint jwt.utils module (#564) by @jdufresne
- Prefer ModuleNotFoundError over ImportError (#565) by @jdufresne
- Fix tox "manifest" environment to pass (#566) by @jdufresne
- Fix tox "docs" environment to pass (#567) by @jdufresne
- Simplify black configuration to be closer to upstream defaults (#568) by @jdufresne
- Use generator expressions (#569) by @jdufresne
- Simplify from_base64url_uint() (#570) by @jdufresne
- Drop lint environment from GitHub actions in favor of pre-commit.ci (#571) by @jdufresne
- [pre-commit.ci] pre-commit autoupdate (#572)
- Simplify tox configuration (#573) by @jdufresne
- Combine identical test functions using pytest.mark.parametrize() (#574) by @jdufresne
- Complete type hinting of jwks_client.py (#578) by @jdufresne

4.6.6 v1.7.1

Fixed

- Update test dependencies with pinned ranges
- Fix pytest deprecation warnings

4.6.7 v1.7.0

Changed

- Remove CRLF line endings #353

Fixed

- Update usage.rst #360

Added

- Support for Python 3.7 #375 #379 #384

4.6.8 v1.6.4

Fixed

- Reverse an unintentional breaking API change to .decode() #352

4.6.9 v1.6.3

Changed

- All exceptions inherit from PyJWTError #340

Added

- Add type hints #344
- Add help module 7ca41e

Docs

- Added section to usage docs for jwt.get_unverified_header() #350
- Update legacy instructions for using pycrypto #337

4.6.10 v1.6.1

Fixed

- Audience parameter throws `InvalidAudienceError` when application does not specify an audience, but the token does. #336

4.6.11 v1.6.0

Changed

- Dropped support for python 2.6 and 3.3 #301
- An invalid signature now raises an `InvalidSignatureError` instead of `DecodeError` #316

Fixed

- Fix over-eager fallback to stdin #304

Added

- Audience parameter now supports iterables #306

4.6.12 v1.5.3

Changed

- Increase required version of the cryptography package to >=1.4.0.

Fixed

- Remove uses of deprecated functions from the cryptography package.
- Warn about missing `algorithms` param to `decode()` only when `verify` param is `True` #281

4.6.13 v1.5.2

Fixed

- Ensure correct arguments order in decode super call 7c1e61d

4.6.14 v1.5.1

Changed

- Change optparse for argparse. #238

Fixed

- Guard against PKCS1 PEM encoded public keys #277
- Add deprecation warning when decoding without specifying `algorithms` #277
- Improve deprecation messages #270
- PyJWT.decode: move verify param into options #271

Added

- Support for Python 3.6 #262
- Expose jwt.InvalidAlgorithmError #264

4.6.15 v1.5.0

Changed

- Add support for ECDSA public keys in RFC 4253 (OpenSSH) format #244
- Renamed commandline script `jwt` to `jwt-cli` to avoid issues with the script clobbering the `jwt` module in some circumstances. #187
- Better error messages when using an algorithm that requires the cryptography package, but it isn't available #230
- Tokens with future 'iat' values are no longer rejected #190
- Non-numeric 'iat' values now raise InvalidIssuedAtError instead of DecodeError
- Remove rejection of future 'iat' claims #252

Fixed

- Add back 'ES512' for backward compatibility (for now) #225
- Fix incorrectly named ECDSA algorithm #219
- Fix rpm build #196

Added

- Add JWK support for HMAC and RSA keys #202

4.6.16 v1.4.2

Fixed

- A PEM-formatted key encoded as bytes could cause a `TypeError` to be raised #213

4.6.17 v1.4.1

Fixed

- Newer versions of Pytest could not detect warnings properly #182
- Non-string 'kid' value now raises `InvalidTokenError` #174
- `jwt.decode(None)` now gracefully fails with `InvalidTokenError` #183

4.6.18 v1.4

Fixed

- Exclude Python cache files from PyPI releases.

Added

- Added new options to require certain claims (require_nbf, require_iat, require_exp) and raise `MissingRequiredClaimError` if they are not present.
- If `audience=` or `issuer=` is specified but the claim is not present, `MissingRequiredClaimError` is now raised instead of `InvalidAudienceError` and `InvalidIssuerError`

4.6.19 v1.3

Fixed

- ECDSA (ES256, ES384, ES512) signatures are now being properly serialized #158
- RSA-PSS (PS256, PS384, PS512) signatures now use the proper salt length for PSS padding. #163

Added

- Added a new `jwt.get_unverified_header()` to parse and return the header portion of a token prior to signature verification.

Removed

- Python 3.2 is no longer a supported platform. This version of Python is rarely used. Users affected by this should upgrade to 3.3+.

4.6.20 v1.2.0

Fixed

- Added back `verify_expiration=` argument to `jwt.decode()` that was erroneously removed in v1.1.0.

Changed

- Refactored JWS-specific logic out of PyJWT and into PyJWS superclass. #141

Deprecated

- `verify_expiration=` argument to `jwt.decode()` is now deprecated and will be removed in a future version. Use the `option=` argument instead.

4.6.21 v1.1.0

Added

- Added support for PS256, PS384, and PS512 algorithms. #132
- Added flexible and complete verification options during decode. #131
- Added this CHANGELOG.md file.

Deprecated

- Deprecated usage of the .decode(..., verify=False) parameter.

Fixed

- Fixed command line encoding. #128

4.6.22 v1.0.1

Fixed

- Include jwt/contrib' and jwt/contrib/algorithms' in setup.py so that they will actually be included when installing. 882524d
- Fix bin/jwt after removing jwt.header(). bd57b02

4.6.23 v1.0.0

Changed

- Moved `jwt.api.header` out of the public API. #85
- Added README details how to extract public / private keys from an x509 certificate. #100
- Refactor api.py functions into an object (`PyJWT`). #101
- Added support for PyCrypto and ecdsa when cryptography isn't available. #101

Fixed

- Fixed a security vulnerability where `alg=None` header could bypass signature verification. #109
- Fixed a security vulnerability by adding support for a whitelist of allowed `alg` values `jwt.decode(algorithms=[])`. #110

Python Module Index

j

Index