
python-jose Documentation

Release 0.2.0

Michael Davis

Jul 22, 2022

CONTENTS

| | | |
|----------|-------------------------------|-----------|
| 1 | Contents | 3 |
| 1.1 | JSON Web Signature | 3 |
| 1.2 | JSON Web Token | 4 |
| 1.3 | JSON Web Key | 4 |
| 1.4 | JSON Web Encryption | 5 |
| 2 | APIs | 7 |
| 2.1 | JWS API | 7 |
| 2.2 | JWT API | 9 |
| 2.3 | JWK API | 11 |
| 2.4 | JWE API | 11 |
| 3 | Principles | 15 |
| 4 | Thanks | 17 |
| | Python Module Index | 19 |
| | Index | 21 |

A JOSE implementation in Python

The JavaScript Object Signing and Encryption (JOSE) technologies - JSON Web Signature (JWS), JSON Web Encryption (JWE), JSON Web Key (JWK), and JSON Web Algorithms (JWA) - collectively can be used to encrypt and/or sign content using a variety of algorithms. While the full set of permutations is extremely large, and might be daunting to some, it is expected that most applications will only use a small set of algorithms to meet their needs.

CONTENTS

1.1 JSON Web Signature

JSON Web Signatures (JWS) are used to digitally sign a JSON encoded object and represent it as a compact URL-safe string.

1.1.1 Supported Algorithms

The following algorithms are currently supported.

| Algorithm Value | Digital Signature or MAC Algorithm |
|-----------------|-------------------------------------|
| HS256 | HMAC using SHA-256 hash algorithm |
| HS384 | HMAC using SHA-384 hash algorithm |
| HS512 | HMAC using SHA-512 hash algorithm |
| RS256 | RSASSA using SHA-256 hash algorithm |
| RS384 | RSASSA using SHA-384 hash algorithm |
| RS512 | RSASSA using SHA-512 hash algorithm |
| ES256 | ECDSA using SHA-256 hash algorithm |
| ES384 | ECDSA using SHA-384 hash algorithm |
| ES512 | ECDSA using SHA-512 hash algorithm |

1.1.2 Examples

Signing tokens

```
>>> from jose import jws
>>> signed = jws.sign({'a': 'b'}, 'secret', algorithm='HS256')
'eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJhIjoiYiJ9.jiMyrsmD8AoHWeQgmzZ5yq8z0lXS67_
  ↳ QGs52AzC8Ru8'
```

Verifying token signatures

```
>>> jws.verify(signed, 'secret', algorithms=['HS256'])
{'a': 'b'}
```

1.2 JSON Web Token

JSON Web Tokens (JWT) are a JWS with a set of reserved claims to be used in a standardized manner.

1.2.1 JWT Reserved Claims

| Claim | Name | Format | Usage |
|-------|------------|------------------|---|
| 'exp' | Expiration | int | The time after which the token is invalid. |
| 'nbf' | Not Before | int | The time before which the token is invalid. |
| 'iss' | Issuer | str | The principal that issued the JWT. |
| 'aud' | Audience | str or list(str) | The recipient that the JWT is intended for. |
| 'iat' | Issued At | int | The time at which the JWT was issued. |

1.3 JSON Web Key

JSON Web Keys (JWK) are a JSON data structure representing a cryptographic key.

1.3.1 Examples

Verifying token signatures

```
>>> from jose import jwk
>>> from jose.utils import base64url_decode
>>>
>>> token =
↳ "eyJhbGciOiJIUzI1NiIsImtpZCI6IjAxOGMwYWU1LTRkOWItNDcxYi1iZmQ2LWVlZjMxNGJjNzAzNyJ9.
↳ SXTigJlzlIGegZGFuZ2Vyb3VzIGJlc2luZXNzLCBGcm9kbywgZ29pbmcgb3V0IHlvdXIgZG9vci4gWW91IHNOZXAgb250byB0aGUgc
↳ s0h6KThzkfBBBkLspW1h84VsJZFTsPPqMDA7g1Md7p0"
>>> hmac_key = {
    "kty": "oct",
    "kid": "018c0ae5-4d9b-471b-bfd6-eef314bc7037",
    "use": "sig",
    "alg": "HS256",
    "k": "hJtXIZ2uSN5kbQfbtTNWbpdmhkV8FJG-Onbc6mxCcYg"
}
>>>
>>> key = jwk.construct(hmac_key)
>>>
>>> message, encoded_sig = token.rsplitt('.', 1)
>>> decoded_sig = base64url_decode(encoded_sig)
>>> key.verify(message, decoded_sig)
```


1.3.2 Note

python-jose requires the use of public keys, as opposed to X.509 certificates. If you have an X.509 certificate that you would like to convert to a public key that python-jose can consume, you can do so with openssl.

```
> openssl x509 -pubkey -noout < cert.pem
```

1.4 JSON Web Encryption

JSON Web Encryption (JWE) are used to encrypt a payload and represent it as a compact URL-safe string.

1.4.1 Supported Content Encryption Algorithms

The following algorithms are currently supported.

| Encryption Value | Encryption Algorithm, Mode, and Auth Tag |
|------------------|--|
| A128CBC_HS256 | AES w/128 bit key in CBC mode w/SHA256 HMAC |
| A192CBC_HS384 | AES w/128 bit key in CBC mode w/SHA256 HMAC |
| A256CBC_HS512 | AES w/128 bit key in CBC mode w/SHA256 HMAC |
| A128GCM | AES w/128 bit key in GCM mode and GCM auth tag |
| A192GCM | AES w/192 bit key in GCM mode and GCM auth tag |
| A256GCM | AES w/256 bit key in GCM mode and GCM auth tag |

1.4.2 Supported Key Management Algorithms

The following algorithms are currently supported.

| Algorithm Value | Key Wrap Algorithm |
|-----------------|--|
| DIR | Direct (no key wrap) |
| RSA1_5 | RSAs with PKCS1 v1.5 |
| RSA_OAEP | RSAs OAEP using default parameters |
| RSA_OAEP_256 | RSAs OAEP using SHA-256 and MGF1 with SHA-256 |
| A128KW | AES Key Wrap with default IV using 128-bit key |
| A192KW m | AES Key Wrap with default IV using 192-bit key |
| A256KW | AES Key Wrap with default IV using 256-bit key |

1.4.3 Examples

Encrypting Payloads

```
>>> from jose import jwe
>>> jwe.encrypt('Hello, World!', 'asecret128bitkey', algorithm='dir', encryption='A128GCM
↪')
'eyJhbGciOiJkaXIiLCJlbmMiOiJBMTI4R0NNIn0..McILMB3dYsNJSuhcDzQshA.OfX9H_mcUpHDeRM4IA.
↪CcnTWqaqxNsjt4eCaUABSg'
```

Decrypting Payloads

```
>>> from jose import jwe
>>> jwe.decrypt('eyJhbGciOiJkaXIiLCJlbmMiOiJBMTI4R0NNIn0..McILMB3dYsNJSuhcDzQshA.OfX9H_
↪mcUpHDeRM4IA.CcnTWqaqxNsjT4eCaUABSg', 'asecret128bitkey')
'Hello, World!'
```

2.1 JWS API

`jose.jws.get_unverified_claims(token)`

Returns the decoded claims without verification of any kind.

Parameters

token (*str*) – A signed JWS to decode the headers from.

Returns

The str representation of the token claims.

Return type

str

Raises

JWSError – If there is an exception decoding the token.

`jose.jws.get_unverified_header(token)`

Returns the decoded headers without verification of any kind.

Parameters

token (*str*) – A signed JWS to decode the headers from.

Returns

The dict representation of the token headers.

Return type

dict

Raises

JWSError – If there is an exception decoding the token.

`jose.jws.get_unverified_headers(token)`

Returns the decoded headers without verification of any kind.

This is simply a wrapper of `get_unverified_header()` for backwards compatibility.

Parameters

token (*str*) – A signed JWS to decode the headers from.

Returns

The dict representation of the token headers.

Return type

dict

Raises

JWSError – If there is an exception decoding the token.

`jose.jws.sign(payload, key, headers=None, algorithm='HS256')`

Signs a claims set and returns a JWS string.

Parameters

- **payload** (*str or dict*) – A string to sign
- **key** (*str or dict*) – The key to use for signing the claim set. Can be individual JWK or JWK set.
- **headers** (*dict, optional*) – A set of headers that will be added to the default headers. Any headers that are added as additional headers will override the default headers.
- **algorithm** (*str, optional*) – The algorithm to use for signing the the claims. Defaults to HS256.

Returns

The string representation of the header, claims, and signature.

Return type

str

Raises

JWSError – If there is an error signing the token.

Examples

```
>>> jws.sign({'a': 'b'}, 'secret', algorithm='HS256')
'eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJhIjoiYiJ9.jiMyrsmD8AoHWeQgmXZ5yq8z0lXS67_
↪ QGs52AzC8Ru8'
```

`jose.jws.verify(token, key, algorithms, verify=True)`

Verifies a JWS string's signature.

Parameters

- **token** (*str*) – A signed JWS to be verified.
- **key** (*str or dict*) – A key to attempt to verify the payload with. Can be individual JWK or JWK set.
- **algorithms** (*str or list*) – Valid algorithms that should be used to verify the JWS.

Returns

The str representation of the payload, assuming the signature is valid.

Return type

str

Raises

JWSError – If there is an exception verifying a token.

Examples

```
>>> token = 'eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJhIjoiYiJ9.
↪jiMyrsmD8AoHWeQgmxZ5yq8z0lXS67_QGs52AzC8Ru8'
>>> jws.verify(token, 'secret', algorithms='HS256')
```

2.2 JWT API

`jose.jwt.decode(token, key, algorithms=None, options=None, audience=None, issuer=None, subject=None, access_token=None)`

Verifies a JWT string's signature and validates reserved claims.

Parameters

- **token** (*str*) – A signed JWS to be verified.
- **key** (*str or dict*) – A key to attempt to verify the payload with. Can be individual JWK or JWK set.
- **algorithms** (*str or list*) – Valid algorithms that should be used to verify the JWS.
- **audience** (*str*) – The intended audience of the token. If the “aud” claim is included in the claim set, then the audience must be included and must equal the provided claim.
- **issuer** (*str or iterable*) – Acceptable value(s) for the issuer of the token. If the “iss” claim is included in the claim set, then the issuer must be given and the claim in the token must be among the acceptable values.
- **subject** (*str*) – The subject of the token. If the “sub” claim is included in the claim set, then the subject must be included and must equal the provided claim.
- **access_token** (*str*) – An access token string. If the “at_hash” claim is included in the claim set, then the access_token must be included, and it must match the “at_hash” claim.
- **options** (*dict*) – A dictionary of options for skipping validation steps.

defaults = {

‘verify_signature’: True, ‘verify_aud’: True, ‘verify_iat’: True, ‘verify_exp’: True, ‘verify_nbf’: True, ‘verify_iss’: True, ‘verify_sub’: True, ‘verify_jti’: True, ‘verify_at_hash’: True, ‘require_aud’: False, ‘require_iat’: False, ‘require_exp’: False, ‘require_nbf’: False, ‘require_iss’: False, ‘require_sub’: False, ‘require_jti’: False, ‘require_at_hash’: False, ‘leeway’: 0,

}

Returns

The dict representation of the claims set, assuming the signature is valid and all requested data validation passes.

Return type

dict

Raises

- **JWTError** – If the signature is invalid in any way.
- **ExpiredSignatureError** – If the signature has expired.
- **JWTClaimsError** – If any claim is invalid in any way.

Examples

```
>>> payload = 'eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJhIjoiYiJ9.  
↪jiMyrsmD8AoHWeQgmxZ5yq8z0lXS67_QGs52AzC8Ru8'  
>>> jwt.decode(payload, 'secret', algorithms='HS256')
```

`jose.jwt.encode(claims, key, algorithm='HS256', headers=None, access_token=None)`

Encodes a claims set and returns a JWT string.

JWTs are JWS signed objects with a few reserved claims.

Parameters

- **claims** (*dict*) – A claims set to sign
- **key** (*str* or *dict*) – The key to use for signing the claim set. Can be individual JWK or JWK set.
- **algorithm** (*str*, *optional*) – The algorithm to use for signing the the claims. Defaults to HS256.
- **headers** (*dict*, *optional*) – A set of headers that will be added to the default headers. Any headers that are added as additional headers will override the default headers.
- **access_token** (*str*, *optional*) – If present, the ‘at_hash’ claim will be calculated and added to the claims present in the ‘claims’ parameter.

Returns

The string representation of the header, claims, and signature.

Return type

str

Raises

JWTError – If there is an error encoding the claims.

Examples

```
>>> jwt.encode({'a': 'b'}, 'secret', algorithm='HS256')  
'eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJhIjoiYiJ9.jiMyrsmD8AoHWeQgmxZ5yq8z0lXS67_  
↪QGs52AzC8Ru8'
```

`jose.jwt.get_unverified_claims(token)`

Returns the decoded claims without verification of any kind.

Parameters

token (*str*) – A signed JWT to decode the headers from.

Returns

The dict representation of the token claims.

Return type

dict

Raises

JWTError – If there is an exception decoding the token.

`jose.jwt.get_unverified_header(token)`

Returns the decoded headers without verification of any kind.

Parameters

token (*str*) – A signed JWT to decode the headers from.

Returns

The dict representation of the token headers.

Return type

dict

Raises

JWTError – If there is an exception decoding the token.

`jose.jwt.get_unverified_headers(token)`

Returns the decoded headers without verification of any kind.

This is simply a wrapper of `get_unverified_header()` for backwards compatibility.

Parameters

token (*str*) – A signed JWT to decode the headers from.

Returns

The dict representation of the token headers.

Return type

dict

Raises

JWTError – If there is an exception decoding the token.

2.3 JWK API

`jose.jwk.construct(key_data, algorithm=None)`

Construct a Key object for the given algorithm with the given `key_data`.

2.4 JWE API

`jose.jwe.decrypt(jwe_str, key)`

Decrypts a JWE compact serialized string and returns the plaintext.

Parameters

- **jwe_str** (*str*) – A JWE to be decrypt.
- **key** (*str* or *dict*) – A key to attempt to decrypt the payload with. Can be individual JWK or JWK set.

Returns

The plaintext bytes, assuming the authentication tag is valid.

Return type

bytes

Raises

JWEError – If there is an exception verifying the token.

Examples

```
>>> from jose import jwe
>>> jwe.decrypt(jwe_string, 'asecret128bitkey')
'Hello, World!'
```

`jose.jwe.encrypt(plaintext, key, encryption='A256GCM', algorithm='dir', zip=None, cty=None, kid=None)`

Encrypts plaintext and returns a JWE compact serialization string.

Parameters

- **plaintext** (*bytes*) – A bytes object to encrypt
- **key** (*str or dict*) – The key(s) to use for encrypting the content. Can be individual JWK or JWK set.
- **encryption** (*str, optional*) – The content encryption algorithm used to perform authenticated encryption on the plaintext to produce the ciphertext and the Authentication Tag. Defaults to A256GCM.
- **algorithm** (*str, optional*) – The cryptographic algorithm used to encrypt or determine the value of the CEK. Defaults to dir.
- **zip** (*str, optional*) – The compression algorithm applied to the plaintext before encryption. Defaults to None.
- **cty** (*str, optional*) – The media type for the secured content. See <http://www.iana.org/assignments/media-types/media-types.xhtml>
- **kid** (*str, optional*) – Key ID for the provided key

Returns

The string representation of the header, encrypted key, initialization vector, ciphertext, and authentication tag.

Return type

bytes

Raises

JWEError – If there is an error signing the token.

Examples

```
>>> from jose import jwe
>>> jwe.encrypt('Hello, World!', 'asecret128bitkey', algorithm='dir', encryption=
↪ 'A128GCM')
'eyJhbGciOiJkaXIiLCJlbmMiOiJBMTI4R0NNIn0..McILMB3dYsNJSuhcDzQshA.OfX9H_mcUpHDeRM4IA.
↪ CcnTWqaqxNsJT4eCaUABSg'
```

`jose.jwe.get_unverified_header(jwe_str)`

Returns the decoded headers without verification of any kind.

Parameters

jwe_str (*str*) – A compact serialized JWE to decode the headers from.

Returns

The dict representation of the JWE headers.

Return type

dict

Raises

JWEError – If there is an exception decoding the JWE.

PRINCIPLES

This is a JOSE implementation that is fully compatible with Google App Engine which requires the use of the PyCrypto library.

CHAPTER
FOUR

THANKS

This library was originally based heavily on the work of the guys over at [PyJWT](#).

PYTHON MODULE INDEX

j

jose.jwe, [11](#)
jose.jwk, [11](#)
jose.jws, [7](#)
jose.jwt, [9](#)

INDEX

C

`construct()` (*in module jose.jwk*), 11

D

`decode()` (*in module jose.jwt*), 9

`decrypt()` (*in module jose.jwe*), 11

E

`encode()` (*in module jose.jwt*), 10

`encrypt()` (*in module jose.jwe*), 12

G

`get_unverified_claims()` (*in module jose.jws*), 7

`get_unverified_claims()` (*in module jose.jwt*), 10

`get_unverified_header()` (*in module jose.jwe*), 12

`get_unverified_header()` (*in module jose.jws*), 7

`get_unverified_header()` (*in module jose.jwt*), 10

`get_unverified_headers()` (*in module jose.jws*), 7

`get_unverified_headers()` (*in module jose.jwt*), 11

J

`jose.jwe`
module, 11

`jose.jwk`
module, 11

`jose.jws`
module, 7

`jose.jwt`
module, 9

M

module
 jose.jwe, 11
 jose.jwk, 11
 jose.jws, 7
 jose.jwt, 9

S

`sign()` (*in module jose.jws*), 8

V

`verify()` (*in module jose.jws*), 8