JWCrypto Documentation

Release 0.8

JWCrypto Contributors

Aug 21, 2020

Contents

1 JSON Web Key (JWK) **3**
- 1.1 Classes . . . 3
- 1.2 Exceptions . . . 6
- 1.3 Registries . . . 6
- 1.4 Examples . . . 7

2 JSON Web Signature (JWS) **9**
- 2.1 Classes . . . 9
- 2.2 Variables . . . 11
- 2.3 Exceptions . . . 11
- 2.4 Registries . . . 12
- 2.5 Examples . . . 12

3 JSON Web Encryption (JWE) **13**
- 3.1 Classes . . . 13
- 3.2 Variables . . . 15
- 3.3 Exceptions . . . 15
- 3.4 Registries . . . 15
- 3.5 Examples . . . 15

4 JSON Web Token (JWT) **17**
- 4.1 Classes . . . 17
- 4.2 Examples . . . 18

5 Indices and tables **21**

Index **23**

JWCrypto is an implementation of the Javascript Object Signing and Encryption (JOSE) Web Standards as they are being developed in the JOSE IETF Working Group and related technology.

JWCrypto is Python2 and Python3 compatible and uses the Cryptography package for all the crypto functions.

Contents:

CHAPTER 1

JSON Web Key (JWK)

The jwk Module implements the JSON Web Key standard. A JSON Web Key is represented by a JWK object, related utility classes and functions are available in this module too.

1.1 Classes

class jwcrypto.jwk.**JWK**(***kwargs*)

Bases: object

JSON Web Key object

This object represent a Key. It must be instantiated by using the standard defined key/value pairs as arguments of the initialization function.

Creates a new JWK object.

The function arguments must be valid parameters as defined in the 'IANA JSON Web Key Set Parameters registry' and specified in the *JWKParamsRegistry* variable. The 'kty' parameter must always be provided and its value must be a valid one as defined by the 'IANA JSON Web Key Types registry' and specified in the *JWKTypesRegistry* variable. The valid key parameters per key type are defined in the *JWKValuesRegistry* variable.

To generate a new random key call the class method generate() with the appropriate 'kty' parameter, and other parameters as needed (key size, public exponents, curve types, etc..)

Valid options per type, when generating new keys:

- oct: size(int)
- RSA: public_exponent(int), size(int)
- EC: crv(str) (one of P-256, P-384, P-521, secp256k1)
- OKP: crv(str) (one of Ed25519, Ed448, X25519, X448)

Deprecated: Alternatively if the 'generate' parameter is provided, with a valid key type as value then a new key will be generated according to the defaults or provided key strength options (type specific).

Raises

- ***InvalidJWKType*** – if the key type is invalid
- ***InvalidJWKValue*** – if incorrect or inconsistent parameters are provided.

export(*private_key=True*, *as_dict=False*)

Exports the key in the standard JSON format. Exports the key regardless of type, if private_key is False and the key is_symmetric an exception is raised.

Parameters **private_key(bool)** – Whether to export the private key. Defaults to True.

export_private(*as_dict=False*)

Export the private key in the standard JSON format. It fails for a JWK that has only a public key or is symmetric.

Parameters **as_dict(bool)** – If set to True export as python dict not JSON

export_public(*as_dict=False*)

Exports the public key in the standard JSON format. It fails if one is not available like when this function is called on a symmetric key.

Parameters **as_dict(bool)** – If set to True export as python dict not JSON

export_to_pem(*private_key=False*, *password=False*)

Exports keys to a data buffer suitable to be stored as a PEM file. Either the public or the private key can be exported to a PEM file. For private keys the PKCS#8 format is used. If a password is provided the best encryption method available as determined by the cryptography module is used to wrap the key.

Parameters

- **private_key** – Whether the private key should be exported. Defaults to *False* which means the public key is exported by default.
- **password(bytes)** – A password for wrapping the private key. Defaults to False which will cause the operation to fail. To avoid encryption the user must explicitly pass None, otherwise the user needs to provide a password in a bytes buffer.

classmethod from_json(*key*)

Creates a RFC 7517 JWK from the standard JSON format.

Parameters **key** – The RFC 7517 representation of a JWK.

classmethod from_pem(*data*, *password=None*)

Creates a key from PKCS#8 formatted data loaded from a PEM file. See the function *import_from_pem* for details.

Parameters

- **data(bytes)** – The data contained in a PEM file.
- **password(bytes)** – An optional password to unwrap the key.

get_curve(*arg*)

Gets the Elliptic Curve associated with the key.

Parameters **arg** – an optional curve name

Raises

- ***InvalidJWKType*** – the key is not an EC or OKP key.
- ***InvalidJWKValue*** – if the curve names is invalid.

get_op_key(*operation=None*, *arg=None*)

Get the key object associated to the requested operation. For example the public RSA key for the 'verify' operation or the private EC key for the 'decrypt' operation.

Parameters

- **operation** – The requested operation. The valid set of operations is available in the *JWKOperationsRegistry* registry.
- **arg** – an optional, context specific, argument For example a curve name.

Raises

- ***InvalidJWKOperation*** – if the operation is unknown or not permitted with this key.
- ***InvalidJWKUsage*** – if the use constraints do not permit the operation.

import_from_pem(*data*, *password=None*)

Imports a key from data loaded from a PEM file. The key may be encrypted with a password. Private keys (PKCS#8 format), public keys, and X509 certificate's public keys can be imported with this interface.

Parameters

- **data(bytes)** – The data contained in a PEM file.
- **password(bytes)** – An optional password to unwrap the key.

thumbprint(*hashalg=<cryptography.hazmat.primitives.hashes.SHA256 object>*)

Returns the key thumbprint as specified by RFC 7638.

Parameters **hashalg** – A hash function (defaults to SHA256)

has_private

Whether this JWK has an asymmetric key Private key.

has_public

Whether this JWK has an asymmetric Public key.

is_symmetric

Whether this JWK is a symmetric key.

key_curve

The Curve Name.

key_id

The Key ID. Provided by the kid parameter if present, otherwise returns None.

key_type

The Key type

class jwcrypto.jwk.**JWKSet**(**args*, ***kwargs*)

Bases: dict

A set of JWK objects.

Inherits from the standard 'dict' builtin type. Creates a special key 'keys' that is of a type derived from 'set' The 'keys' attribute accepts only *jwcrypto.jwk.JWK* elements.

export(*private_keys=True*, *as_dict=False*)

Exports a RFC 7517 key set. Exports as json by default, or as dict if requested.

Parameters

- **private_key(bool)** – Whether to export private keys. Defaults to True.

- **as_dict(bool)** – Whether to return a dict instead of a JSON object

classmethod from_json(*keyset*)
Creates a RFC 7517 key set from the standard JSON format.

Parameters **keyset** – The RFC 7517 representation of a JOSE key set.

get_key(*kid*)
Gets a key from the set. :param kid: the 'kid' key identifier.

import_keyset(*keyset*)
Imports a RFC 7517 key set using the standard JSON format.

Parameters **keyset** – The RFC 7517 representation of a JOSE key set.

update([*E*], ***F*) → None. Update D from dict/iterable E and F.
If E present and has a .keys() method, does: for k in E: D[k] = E[k] If E present and lacks .keys() method, does: for (k, v) in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

1.2 Exceptions

class jwcrypto.jwk.**InvalidJWKType**(*value=None*)
Bases: jwcrypto.common.JWException

Invalid JWK Type Exception.

This exception is raised when an invalid parameter type is used.

class jwcrypto.jwk.**InvalidJWKValue**
Bases: jwcrypto.common.JWException

Invalid JWK Value Exception.

This exception is raised when an invalid/unknown value is used in the context of an operation that requires specific values to be used based on the key type or other constraints.

class jwcrypto.jwk.**InvalidJWKOperation**(*operation*, *values*)
Bases: jwcrypto.common.JWException

Invalid JWK Operation Exception.

This exception is raised when an invalid key operation is requested, based on the key type and declared usage constraints.

class jwcrypto.jwk.**InvalidJWKUsage**(*use*, *value*)
Bases: jwcrypto.common.JWException

Invalid JWK usage Exception.

This exception is raised when an invalid key usage is requested, based on the key type and declared usage constraints.

1.3 Registries

jwcrypto.jwk.**JWKTypesRegistry**
Registry of valid Key Types

jwcrypto.jwk.**JWKValuesRegistry**
Registry of valid key values

`jwcrypto.jwk.`**`JWKParamsRegistry`**
Registry of valid key parameters

`jwcrypto.jwk.`**`JWKEllipticCurveRegistry`**
Registry of allowed Elliptic Curves

`jwcrypto.jwk.`**`JWKUseRegistry`**
Registry of allowed uses

`jwcrypto.jwk.`**`JWKOperationsRegistry`**
Registry of allowed operations

1.4 Examples

Create a 256bit symmetric key::

```
>>> from jwcrypto import jwk
>>> key = jwk.JWK.generate(kty='oct', size=256)
```

Export the key with::

```
>>> key.export()
'{"k":"X6TBlwY2so8EwKZ2TFXM7XHSgWBKQJhcspzYydp5Y-o","kty":"oct"}'
```

Create a 2048bit RSA key pair::

```
>>> jwk.JWK.generate(kty='RSA', size=2048)
```

Create a P-256 EC key pair and export the public key::

```
>>> key = jwk.JWK.generate(kty='EC', crv='P-256')
>>> key.export(private_key=False)
'{"y":"VYlYwBfOTIICojCPfdUjnmkpN-g-lzZKxzjAoFmDRm8",
  "x":"3mdE0rODWRju6qqU01Kw5oPYdNxBOMisFvJFH1vEu9Q",
  "crv":"P-256","kty":"EC"}'
```

Import a P-256 Public Key::

```
>>> expkey = {"y":"VYlYwBfOTIICojCPfdUjnmkpN-g-lzZKxzjAoFmDRm8",
              "x":"3mdE0rODWRju6qqU01Kw5oPYdNxBOMisFvJFH1vEu9Q",
              "crv":"P-256","kty":"EC"}
>>> key = jwk.JWK(**expkey)
```

Import a Key from a PEM file::

```
>>> with open("public.pem", "rb") as pemfile:
>>>     key = jwk.JWK.from_pem(pemfile.read())
```

CHAPTER 2

JSON Web Signature (JWS)

The jws Module implements the JSON Web Signature standard. A JSON Web Signature is represented by a JWS object, related utility classes and functions are available in this module too.

2.1 Classes

class `jwcrypto.jws.`**`JWS`**(*payload=None*, *header_registry=None*)

Bases: `object`

JSON Web Signature object

This object represent a JWS token.

Creates a JWS object.

Parameters

- **`payload(bytes)`** – An arbitrary value (optional).
- **`header_registry`** – Optional additions to the header registry

`add_signature`(*key*, *alg=None*, *protected=None*, *header=None*)

Adds a new signature to the object.

Parameters

- **`key`** – A (`jwcrypto.jwk.JWK`) key of appropriate for the "alg" provided.
- **`alg`** – An optional algorithm name. If already provided as an element of the protected or unprotected header it can be safely omitted.
- **`protected`** – The Protected Header (optional)
- **`header`** – The Unprotected Header (optional)

Raises

- **InvalidJWSObject** – if no payload has been set on the object, or invalid headers are provided.
- **ValueError** – if the key is not a `JWK` object.
- **ValueError** – if the algorithm is missing or is not provided by one of the headers.
- **InvalidJWAAlgorithm** – if the algorithm is not valid, is unknown or otherwise not yet implemented.

deserialize(*raw_jws*, *key=None*, *alg=None*)

Deserialize a JWS token.

NOTE: Destroys any current status and tries to import the raw JWS provided.

Parameters

- **raw_jws** – a 'raw' JWS token (JSON Encoded or Compact notation) string.
- **key** – A (*jwcrypto.jwk.JWK*) verification key (optional). If a key is provided a verification step will be attempted after the object is successfully deserialized.
- **alg** – The signing algorithm (optional). usually the algorithm is known as it is provided with the JOSE Headers of the token.

Raises

- **InvalidJWSObject** – if the raw object is an invalid JWS token.
- **InvalidJWSSignature** – if the verification fails.

serialize(*compact=False*)

Serializes the object into a JWS token.

Parameters **compact (boolean)** – if True generates the compact representation, otherwise generates a standard JSON format.

Raises

- **InvalidJWSOperation** – if the object cannot serialized with the compact representation and *compact* is True.
- **InvalidJWSSignature** – if no signature has been added to the object, or no valid signature can be found.

verify(*key*, *alg=None*)

Verifies a JWS token.

Parameters

- **key** – The (*jwcrypto.jwk.JWK*) verification key.
- **alg** – The signing algorithm (optional). usually the algorithm is known as it is provided with the JOSE Headers of the token.

Raises **InvalidJWSSignature** – if the verification fails.

allowed_algs

Allowed algorithms.

The list of allowed algorithms. Can be changed by setting a list of algorithm names.

class jwcrypto.jws.**JWSCore**(*alg*, *key*, *header*, *payload*, *algs=None*)

Bases: `object`

The inner JWS Core object.

This object SHOULD NOT be used directly, the JWS object should be used instead as JWS perform necessary checks on the validity of the object and requested operations.

Core JWS token handling.

Parameters

- **alg** – The algorithm used to produce the signature. See RFC 7518
- **key** – A (*jwcrypto.jwk.JWK*) key of appropriate type for the "alg" provided in the 'protected' json string.
- **header** – A JSON string representing the protected header.
- **payload(bytes)** – An arbitrary value
- **algs** – An optional list of allowed algorithms

Raises

- **ValueError** – if the key is not a `JWK` object
- **InvalidJWAAlgorithm** – if the algorithm is not valid, is unknown or otherwise not yet implemented.

sign()
Generates a signature

verify(*signature*)
Verifies a signature

Raises *InvalidJWSSignature* – if the verification fails.

2.2 Variables

jwcrypto.jws.**default_allowed_algs = ['HS256', 'HS384', 'HS512', 'RS256', 'RS384', 'RS512',**
Default allowed algorithms

2.3 Exceptions

class jwcrypto.jws.**InvalidJWSSignature**(*message=None*, *exception=None*)
Bases: `jwcrypto.common.JWException`

Invalid JWS Signature.

This exception is raised when a signature cannot be validated.

class jwcrypto.jws.**InvalidJWSObject**(*message=None*, *exception=None*)
Bases: `jwcrypto.common.JWException`

Invalid JWS Object.

This exception is raised when the JWS Object is invalid and/or improperly formatted.

class jwcrypto.jws.**InvalidJWSOperation**(*message=None*, *exception=None*)
Bases: `jwcrypto.common.JWException`

Invalid JWS Object.

This exception is raised when a requested operation cannot be execute due to unsatisfied conditions.

2.4 Registries

`jwcrypto.jws.`**`JWSHeaderRegistry`**
Registry of valid header parameters

2.5 Examples

Sign a JWS token::

```
>>> from jwcrypto import jwk, jws
>>> from jwcrypto.common import json_encode
>>> key = jwk.JWK.generate(kty='oct', size=256)
>>> payload = "My Integrity protected message"
>>> jwstoken = jws.JWS(payload.encode('utf-8'))
>>> jwstoken.add_signature(key, None,
                           json_encode({"alg": "HS256"}),
                           json_encode({"kid": key.thumbprint()}))
>>> sig = jwstoken.serialize()
```

Verify a JWS token::

```
>>> jwstoken = jws.JWS()
>>> jwstoken.deserialize(sig)
>>> jwstoken.verify(key)
>>> payload = jwstoken.payload
```

CHAPTER 3

JSON Web Encryption (JWE)

The jwe Module implements the JSON Web Encryption standard. A JSON Web Encryption is represented by a JWE object, related utility classes and functions are available in this module too.

3.1 Classes

class `jwcrypto.jwe.`**`JWE`**(*plaintext=None*, *protected=None*, *unprotected=None*, *aad=None*, *algs=None*, *recipient=None*, *header=None*, *header_registry=None*)

Bases: `object`

JSON Web Encryption object

This object represent a JWE token.

Creates a JWE token.

Parameters

- **`plaintext(bytes)`** – An arbitrary plaintext to be encrypted.
- **`protected`** – A JSON string with the protected header.
- **`unprotected`** – A JSON string with the shared unprotected header.
- **`aad(bytes)`** – Arbitrary additional authenticated data
- **`algs`** – An optional list of allowed algorithms
- **`recipient`** – An optional, default recipient key
- **`header`** – An optional header for the default recipient
- **`header_registry`** – Optional additions to the header registry

`add_recipient`(*key*, *header=None*)

Encrypt the plaintext with the given key.

Parameters

- **key** – A JWK key or password of appropriate type for the 'alg' provided in the JOSE Headers.
- **header** – A JSON string representing the per-recipient header.

Raises

- **ValueError** – if the plaintext is missing or not of type bytes.
- **ValueError** – if the compression type is unknown.
- **InvalidJWAAlgorithm** – if the 'alg' provided in the JOSE headers is missing or unknown, or otherwise not implemented.

decrypt(*key*)

Decrypt a JWE token.

Parameters

- **key** – The (*jwcrypto.jwk.JWK*) decryption key.
- **key** – A (*jwcrypto.jwk.JWK*) decryption key or a password string (optional).

Raises

- ***InvalidJWEOperation*** – if the key is not a JWK object.
- ***InvalidJWEData*** – if the ciphertext can't be decrypted or the object is otherwise malformed.

deserialize(*raw_jwe*, *key=None*)

Deserialize a JWE token.

NOTE: Destroys any current status and tries to import the raw JWE provided.

Parameters

- **raw_jwe** – a 'raw' JWE token (JSON Encoded or Compact notation) string.
- **key** – A (*jwcrypto.jwk.JWK*) decryption key or a password string (optional). If a key is provided a decryption step will be attempted after the object is successfully deserialized.

Raises

- ***InvalidJWEData*** – if the raw object is an invalid JWE token.
- ***InvalidJWEOperation*** – if the decryption fails.

serialize(*compact=False*)

Serializes the object into a JWE token.

Parameters **compact(boolean)** – if True generates the compact representation, otherwise generates a standard JSON format.

Raises

- ***InvalidJWEOperation*** – if the object cannot serialized with the compact representation and *compact* is True.
- ***InvalidJWEOperation*** – if no recipients have been added to the object.

allowed_algs

Allowed algorithms.

The list of allowed algorithms. Can be changed by setting a list of algorithm names.

3.2 Variables

`jwcrypto.jwe.`**`default_allowed_algs = ['RSA1_5', 'RSA-OAEP', 'RSA-OAEP-256', 'A128KW', 'A192`**

Default allowed algorithms

3.3 Exceptions

class `jwcrypto.jwe.`**`InvalidJWEOperation`**(*message=None*, *exception=None*)

Bases: `jwcrypto.common.JWException`

Invalid JWS Object.

This exception is raised when a requested operation cannot be execute due to unsatisfied conditions.

class `jwcrypto.jwe.`**`InvalidJWEData`**(*message=None*, *exception=None*)

Bases: `jwcrypto.common.JWException`

Invalid JWE Object.

This exception is raised when the JWE Object is invalid and/or improperly formatted.

class `jwcrypto.jwe.`**`InvalidJWEKeyType`**(*expected*, *obtained*)

Bases: `jwcrypto.common.JWException`

Invalid JWE Key Type.

This exception is raised when the provided JWK Key does not match the type required by the specified algorithm.

class `jwcrypto.jwe.`**`InvalidJWEKeyLength`**(*expected*, *obtained*)

Bases: `jwcrypto.common.JWException`

Invalid JWE Key Length.

This exception is raised when the provided JWK Key does not match the length required by the specified algorithm.

class `jwcrypto.jwe.`**`InvalidCEKeyLength`**(*expected*, *obtained*)

Bases: `jwcrypto.common.JWException`

Invalid CEK Key Length.

This exception is raised when a Content Encryption Key does not match the required length.

3.4 Registries

`jwcrypto.jwe.`**`JWEHeaderRegistry`**

Registry of valid header parameters

3.5 Examples

3.5.1 Symmetric keys

Encrypt a JWE token::

```
>>> from jwcrypto import jwk, jwe
>>> from jwcrypto.common import json_encode
>>> key = jwk.JWK.generate(kty='oct', size=256)
>>> payload = "My Encrypted message"
>>> jwetoken = jwe.JWE(payload.encode('utf-8'),
                       json_encode({"alg": "A256KW",
                                    "enc": "A256CBC-HS512"}))
>>> jwetoken.add_recipient(key)
>>> enc = jwetoken.serialize()
```

Decrypt a JWE token::

```
>>> jwetoken = jwe.JWE()
>>> jwetoken.deserialize(enc)
>>> jwetoken.decrypt(key)
>>> payload = jwetoken.payload
```

3.5.2 Asymmetric keys

Encrypt a JWE token::

```
>>> from jwcrypto import jwk, jwe
>>> from jwcrypto.common import json_encode, json_decode
>>> public_key = jwk.JWK()
>>> private_key = jwk.JWK.generate(kty='RSA', size=2048)
>>> public_key.import_key(**json_decode(private_key.export_public()))
>>> payload = "My Encrypted message"
>>> protected_header = {
        "alg": "RSA-OAEP-256",
        "enc": "A256CBC-HS512",
        "typ": "JWE",
        "kid": public_key.thumbprint(),
    }
>>> jwetoken = jwe.JWE(payload.encode('utf-8'),
                       recipient=public_key,
                       protected=protected_header)
>>> enc = jwetoken.serialize()
```

Decrypt a JWE token::

```
>>> jwetoken = jwe.JWE()
>>> jwetoken.deserialize(enc, key=private_key)
>>> payload = jwetoken.payload
```

CHAPTER 4

JSON Web Token (JWT)

The jwt Module implements the JSON Web Token standard. A JSON Web Token is represented by a JWT object, related utility classes and functions are available in this module too.

4.1 Classes

class jwcrypto.jwt.**JWT**(*header=None*, *claims=None*, *jwt=None*, *key=None*, *algs=None*, *default_claims=None*, *check_claims=None*)

Bases: object

JSON Web token object

This object represent a generic token.

Creates a JWT object.

Parameters

- **header** – A dict or a JSON string with the JWT Header data.
- **claims** – A dict or a string with the JWT Claims data.
- **jwt** – a 'raw' JWT token
- **key** – A (*jwcrypto.jwk.JWK*) key to deserialize the token. A (*jwcrypto.jwk.JWKSet*) can also be used.
- **algs** – An optional list of allowed algorithms
- **default_claims** – An optional dict with default values for registered claims. A None value for NumericDate type claims will cause generation according to system time. Only the values from RFC 7519 - 4.1 are evaluated.
- **check_claims** – An optional dict of claims that must be present in the token, if the value is not None the claim must match exactly.

Note: either the header,claims or jwt,key parameters should be provided as a deserialization operation (which occurs if the jwt is provided) will wipe any header or claim provided by setting those obtained from the deserialization of the jwt token.

Note: if check_claims is not provided the 'exp' and 'nbf' claims are checked if they are set on the token but not enforced if not set. Any other RFC 7519 registered claims are checked only for format conformance.

deserialize (*jwt*, *key=None*)
Deserialize a JWT token.

NOTE: Destroys any current status and tries to import the raw token provided.

Parameters

- **jwt** – a 'raw' JWT token.
- **key** – A (*jwcrypto.jwk.JWK*) verification or decryption key, or a (*jwcrypto.jwk.JWKSet*) that contains a key indexed by the 'kid' header.

make_encrypted_token (*key*)
Encrypts the payload.

Creates a JWE token with the header as the JWE protected header and the claims as the plaintext. See (*jwcrypto.jwe.JWE*) for details on the exceptions that may be raised.

Parameters **key** – A (*jwcrypto.jwk.JWK*) key.

make_signed_token (*key*)
Signs the payload.

Creates a JWS token with the header as the JWS protected header and the claims as the payload. See (*jwcrypto.jws.JWS*) for details on the exceptions that may be raised.

Parameters **key** – A (*jwcrypto.jwk.JWK*) key.

serialize (*compact=True*)
Serializes the object into a JWS token.

Parameters **compact (boolean)** – must be True.

Note: the compact parameter is provided for general compatibility with the serialize() functions of *jwcrypto.jws.JWS* and *jwcrypto.jwe.JWE* so that these objects can all be used interchangeably. However the only valid JWT representation is the compact representation.

4.2 Examples

Create a symmetric key::

```
>>> from jwcrypto import jwt, jwk
>>> key = jwk.JWK(generate='oct', size=256)
>>> key.export()
'{"k":"Wal4ZHCBsml0Al_Y8faoNTKsXCkw8eefKXYFuwTBOpA","kty":"oct"}'
```

Create a signed token with the generated key::

```
>>> Token = jwt.JWT(header={"alg": "HS256"},
                    claims={"info": "I'm a signed token"})
>>> Token.make_signed_token(key)
>>> Token.serialize()
u'eyJhbGciOiJIUzI1NiJ9.eyJpbmZvIjoiSSdtIGEgc2lnbmVkIHRva2VuIn0.
↪rjnRMAKcaRamEHnENhg0_Fqv7Obo-30U4bcI_v-nfEM'
```

Further encrypt the token with the same key::

```
>>> Etoken = jwt.JWT(header={"alg": "A256KW", "enc": "A256CBC-HS512"},
                     claims=Token.serialize())
>>> Etoken.make_encrypted_token(key)
>>> Etoken.serialize()
u'eyJhbGciOiJBMjU2S1ciLCJlbmMiOiJBMjU2Q0JDLUhTNTEyIn0.
↪ST5RmjqDLj696xo7YFTFuKUhcd3naCrm6yMjBM3cqWiFD6U8j2JIsbclsF7ryNg8Ktmt1kQJRKavV6DaTl1T840tP3sIs1
↪wSxVhZH5GyzbJnPBAUMdzQ.6uiVYwrRBzAm7Uge9rEUjExPWGbgerF177A7tMuQurJAqBhgk3_
↪5vee5DRH84kHSapFOxcEuDdMBEQLI7V2E0F57-d01TFStHzwtgtSmeZRQ6JSIL5XlgJouwHfSxn9Z_
↪TGl5xxq4TksORHED1vnRA.5jPyPWanJVqlOohApEbHmxi3JHp1MXbmvQe2_dVd8FI'
```

Now decrypt and verify::

```
>>> from jwcrypto import jwt, jwk
>>> k = {"k": "Wal4ZHCBsml0Al_Y8faoNTKsXCkw8eefKXYFuwTBOpA", "kty": "oct"}
>>> key = jwk.JWK(**k)
>>> e = u'eyJhbGciOiJBMjU2S1ciLCJlbmMiOiJBMjU2Q0JDLUhTNTEyIn0.
↪ST5RmjqDLj696xo7YFTFuKUhcd3naCrm6yMjBM3cqWiFD6U8j2JIsbclsF7ryNg8Ktmt1kQJRKavV6DaTl1T840tP3sIs1
↪wSxVhZH5GyzbJnPBAUMdzQ.6uiVYwrRBzAm7Uge9rEUjExPWGbgerF177A7tMuQurJAqBhgk3_
↪5vee5DRH84kHSapFOxcEuDdMBEQLI7V2E0F57-d01TFStHzwtgtSmeZRQ6JSIL5XlgJouwHfSxn9Z_
↪TGl5xxq4TksORHED1vnRA.5jPyPWanJVqlOohApEbHmxi3JHp1MXbmvQe2_dVd8FI'
>>> ET = jwt.JWT(key=key, jwt=e)
>>> ST = jwt.JWT(key=key, jwt=ET.claims)
>>> ST.claims
u'{"info":"I\'m a signed token"}'
```

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

Index

A

add_recipient() (*jwcrypto.jwe.JWE method*), 13
add_signature() (*jwcrypto.jws.JWS method*), 9
allowed_algs (*jwcrypto.jwe.JWE attribute*), 14
allowed_algs (*jwcrypto.jws.JWS attribute*), 10

D

decrypt() (*jwcrypto.jwe.JWE method*), 14
default_allowed_algs (*in module jwcrypto.jwe*), 15
default_allowed_algs (*in module jwcrypto.jws*), 11
deserialize() (*jwcrypto.jwe.JWE method*), 14
deserialize() (*jwcrypto.jws.JWS method*), 10
deserialize() (*jwcrypto.jwt.JWT method*), 18

E

export() (*jwcrypto.jwk.JWK method*), 4
export() (*jwcrypto.jwk.JWKSet method*), 5
export_private() (*jwcrypto.jwk.JWK method*), 4
export_public() (*jwcrypto.jwk.JWK method*), 4
export_to_pem() (*jwcrypto.jwk.JWK method*), 4

F

from_json() (*jwcrypto.jwk.JWK class method*), 4
from_json() (*jwcrypto.jwk.JWKSet class method*), 6
from_pem() (*jwcrypto.jwk.JWK class method*), 4

G

get_curve() (*jwcrypto.jwk.JWK method*), 4
get_key() (*jwcrypto.jwk.JWKSet method*), 6
get_op_key() (*jwcrypto.jwk.JWK method*), 4

H

has_private (*jwcrypto.jwk.JWK attribute*), 5
has_public (*jwcrypto.jwk.JWK attribute*), 5

I

import_from_pem() (*jwcrypto.jwk.JWK method*), 5
import_keyset() (*jwcrypto.jwk.JWKSet method*), 6
InvalidCEKeyLength (*class in jwcrypto.jwe*), 15
InvalidJWEData (*class in jwcrypto.jwe*), 15
InvalidJWEKeyLength (*class in jwcrypto.jwe*), 15
InvalidJWEKeyType (*class in jwcrypto.jwe*), 15
InvalidJWEOperation (*class in jwcrypto.jwe*), 15
InvalidJWKOperation (*class in jwcrypto.jwk*), 6
InvalidJWKType (*class in jwcrypto.jwk*), 6
InvalidJWKUsage (*class in jwcrypto.jwk*), 6
InvalidJWKValue (*class in jwcrypto.jwk*), 6
InvalidJWSObject (*class in jwcrypto.jws*), 11
InvalidJWSOperation (*class in jwcrypto.jws*), 11
InvalidJWSSignature (*class in jwcrypto.jws*), 11
is_symmetric (*jwcrypto.jwk.JWK attribute*), 5

J

JWE (*class in jwcrypto.jwe*), 13
JWEHeaderRegistry (*in module jwcrypto.jwe*), 15
JWK (*class in jwcrypto.jwk*), 3
JWKEllipticCurveRegistry (*in module jwcrypto.jwk*), 7
JWKOperationsRegistry (*in module jwcrypto.jwk*), 7
JWKParamsRegistry (*in module jwcrypto.jwk*), 6
JWKSet (*class in jwcrypto.jwk*), 5
JWKTypesRegistry (*in module jwcrypto.jwk*), 6
JWKUseRegistry (*in module jwcrypto.jwk*), 7
JWKValuesRegistry (*in module jwcrypto.jwk*), 6
JWS (*class in jwcrypto.jws*), 9
JWSCore (*class in jwcrypto.jws*), 10
JWSHeaderRegistry (*in module jwcrypto.jws*), 12
JWT (*class in jwcrypto.jwt*), 17

K

key_curve (*jwcrypto.jwk.JWK attribute*), 5
key_id (*jwcrypto.jwk.JWK attribute*), 5
key_type (*jwcrypto.jwk.JWK attribute*), 5

M

make_encrypted_token() (*jwcrypto.jwt.JWT*

method), 18
`make_signed_token()` (*jwcrypto.jwt.JWT method*), 18

S

`serialize()` (*jwcrypto.jwe.JWE method*), 14
`serialize()` (*jwcrypto.jws.JWS method*), 10
`serialize()` (*jwcrypto.jwt.JWT method*), 18
`sign()` (*jwcrypto.jws.JWSCore method*), 11

T

`thumbprint()` (*jwcrypto.jwk.JWK method*), 5

U

`update()` (*jwcrypto.jwk.JWKSet method*), 6

V

`verify()` (*jwcrypto.jws.JWS method*), 10
`verify()` (*jwcrypto.jws.JWSCore method*), 11