Python CWT

Release v0.2.0

AJITOMI Daisuke

May 04, 2021

CONTENTS

Python CWT is a CBOR Web Token (CWT) and CBOR Object Signing and Encryption (COSE) implementation compliant with:

- RFC8392: CBOR Web Token (CWT)
- RFC8152: CBOR Object Signing and Encryption (COSE)

It is designed to make users who already know about JSON Web Token (JWT) be able to use it in ease. Little knowledge of CBOR, COSE, and CWT is required to use it.

You can install Python CWT with pip:

```
$ pip install cwt
```

And then, you can use it as follows:

```
import cwt
from cwt import cose_key

key = cose_key.from_symmetric_key(alg="HMAC 256/256")
token = cwt.encode({"iss": "https://as.example", "sub": "dajiaji", "cti": "123"}, key)
decoded = cwt.decode(token, key)
```

CHAPTER

ONE

INDEX

1.1 Installation

You can install Python CWT with pip:

```
$ pip install cwt
```

1.2 Usage Examples

The following is the simplest sample code.

```
>>> import cwt
>>> from cwt import claims, cose_key
>>> key = cose_key.from_symmetric_key(alg="HS256")
>>> token = cwt.encode({"iss": "coaps://as.example", "sub": "dajiaji", "cti": "123"},␣
↪key)
>>> token.hex()
'd18443a10105a05835a60172636f6170733a2f2f61732e6578616d706c65026764616a69616a69'
'0743313233041a609097b7051a609089a7061a609089a758201fad9b0a76803194bd11ca9b9b3c'
'bbf1028005e15321665a768994f38c7127f7'
>>> decoded = cwt.decode(token, key)
>>> decoded
{1: 'coaps://as.example', 2: 'dajiaji', 7: b'123',
 4: 1620088759, 5: 1620085159, 6: 1620085159}
>>> readable = claims.from_dict(decoded)
>>> readable.iss
'coaps://as.example'
>>> readable.sub
'dajiaji'
>>> readable.exp
1620088759
```

This page shows various examples to use this library. Specific examples are as follows:

- *MACed CWT*
- *Signed CWT*
- *Encrypted CWT*
- *Nested CWT*

- *CWT with User-Defined Claims*
- *CWT with PoP key*

1.2.1 MACed CWT

Create a MACed CWT, verify and decode it as follows:

```
import cwt
from cwt import cose_key

try:
    key = cose_key.from_symmetric_key(alg="HS256")
    token = cwt.encode(
        {"iss": "coaps://as.example", "sub": "dajiaji", "cti": "123"},
        key,
    )
    decoded = cwt.decode(token, key)
except Exception as err:
    # All the other examples in this document omit error handling but this CWT library
    # can throw following errors:
    #   ValueError: Invalid arguments.
    #   EncodeError: Failed to encode.
    #   VerifyError: Failed to verify.
    #   DecodeError: Failed to decode.
    print(err)
```

CBOR-like structure (Dict[int, Any]) can also be used as follows:

```
import cwt
from cwt import cose_key

key = cose_key.from_symmetric_key(alg="HS256")
token = cwt.encode({1: "coaps://as.example", 2: "dajiaji", 7: b"123"}, key)
decoded = cwt.decode(token, key)
```

Algorithms other than `HS256` are listed in Supported COSE Algorithms .

1.2.2 Signed CWT

Create an `Ed25519` (Ed25519 for use w/ EdDSA only) key pair:

```
$ openssl genpkey -algorithm ed25519 -out private_key.pem
$ openssl pkey -in private_key.pem -pubout -out public_key.pem
```

Create a Signed CWT, verify and decode it with the key pair as follows:

```
import cwt
from cwt import cose_key

with open("./private_key.pem") as key_file:
    private_key = cose_key.from_pem(key_file.read(), kid="01")
with open("./public_key.pem") as key_file:
    public_key = cose_key.from_pem(key_file.read(), kid="01")
```

(continues on next page)

(continued from previous page)

```
token = cwt.encode(
    {"iss": "coaps://as.example", "sub": "dajiaji", "cti": "123"}, private_key
)

decoded = cwt.decode(token, public_key)
```

JWKs can also be used instead of the PEM-formatted keys as follows:

```
import cwt
from cwt import cose_key

private_key = cose_key.from_jwk(
    {
        "kty": "OKP",
        "d": "L8JS08VsFZoZxGa9JvzYmCWOwg7zaKcei3KZmYsj7dc",
        "use": "sig",
        "crv": "Ed25519",
        "kid": "01",
        "x": "2E6dX83gqD_D0eAmqnaHe1TC1xuld6iAKXfw2OVATr0",
        "alg": "EdDSA",
    }
)
public_key = cose_key.from_jwk(
    {
        "kty": "OKP",
        "use": "sig",
        "crv": "Ed25519",
        "kid": "01",
        "x": "2E6dX83gqD_D0eAmqnaHe1TC1xuld6iAKXfw2OVATr0",
    }
)

token = cwt.encode(
    {"iss": "coaps://as.example", "sub": "dajiaji", "cti": "123"}, private_key
)
decoded = cwt.decode(token, public_key)
```

Algorithms other than `Ed25519` are also supported. The following is an example of `ES256`:

```
$ openssl ecparam -genkey -name prime256v1 -noout -out private_key.pem
$ openssl ec -in private_key.pem -pubout -out public_key.pem
```

```
import cwt
from cwt import cose_key

with open("./private_key.pem") as key_file:
    private_key = cose_key.from_pem(key_file.read(), kid="01")
with open("./public_key.pem") as key_file:
    public_key = cose_key.from_pem(key_file.read(), kid="01")

token = cwt.encode(
    {"iss": "coaps://as.example", "sub": "dajiaji", "cti": "123"}, private_key
)

decoded = cwt.decode(token, public_key)
```

Other supported algorithms are listed in Supported COSE Algorithms.

1.2.3 Encrypted CWT

Create an encrypted CWT with `ChaCha20/Poly1305` (ChaCha20/Poly1305 w/ 256-bit key, 128-bit tag), and decrypt it as follows:

```
import cwt
from cwt import cose_key

enc_key = cose_key.from_symmetric_key(alg="ChaCha20/Poly1305")
token = cwt.encode(
    {"iss": "coaps://as.example", "sub": "dajiaji", "cti": "123"}, enc_key
)
decoded = cwt.decode(token, enc_key)
```

Algorithms other than `ChaCha20/Poly1305` are also supported. The following is an example of `AES-CCM-16-64-256`:

```
import cwt
from cwt import cose_key

enc_key = cose_key.from_symmetric_key(alg="AES-CCM-16-64-256")
token = cwt.encode(
    {"iss": "coaps://as.example", "sub": "dajiaji", "cti": "123"}, enc_key
)
decoded = cwt.decode(token, enc_key)
```

Other supported algorithms are listed in Supported COSE Algorithms.

1.2.4 Nested CWT

Create a signed CWT and encrypt it, and then decrypt and verify the nested CWT as follows.

```
import cwt
from cwt import cose_key

with open("./private_key.pem") as key_file:
    private_key = cose_key.from_pem(key_file.read(), kid="01")
with open("./public_key.pem") as key_file:
    public_key = cose_key.from_pem(key_file.read(), kid="01")

# Creates a CWT with ES256 signing.
token = cwt.encode(
    {"iss": "coaps://as.example", "sub": "dajiaji", "cti": "123"}, private_key
)

# Encrypts the signed CWT.
enc_key = cose_key.from_symmetric_key(alg="ChaCha20/Poly1305")
nested = cwt.encode(token, enc_key)

# Decrypts and verifies the nested CWT.
decoded = cwt.decode(nested, [enc_key, public_key])
```

1.2.5 CWT with User-Defined Claims

You can use your own claims as follows:

Note that such user-defined claim's key should be less than -65536.

```
import cwt
from cwt import cose_key

with open("./private_key.pem") as key_file:
    private_key = cose_key.from_pem(key_file.read(), kid="01")
with open("./public_key.pem") as key_file:
    public_key = cose_key.from_pem(key_file.read(), kid="01")
token = cwt.encode(
    {
        1: "coaps://as.example",  # iss
        2: "dajiaji",  # sub
        7: b"123",  # cti
        -70001: "foo",
        -70002: ["bar"],
        -70003: {"baz": "qux"},
        -70004: 123,
    },
    private_key,
)
raw = cwt.decode(token, public_key)
# raw[-70001] == "foo"
# raw[-70002][0] == "bar"
# raw[-70003]["baz"] == "qux"
# raw[-70004] == 123
readable = claims.from_dict(raw)
# readable.get(-70001) == "foo"
# readable.get(-70002)[0] == "bar"
# readable.get(-70003)["baz"] == "qux"
# readable.get(-70004) == 123
```

User-defined claims can also be used with JSON-based claims as follows:

```
import cwt
from cwt import claims, cose_key

with open("./private_key.pem") as key_file:
    private_key = cose_key.from_pem(key_file.read(), kid="01")
with open("./public_key.pem") as key_file:
    public_key = cose_key.from_pem(key_file.read(), kid="01")

cwt.set_private_claim_names(
    {
        "ext_1": -70001,
        "ext_2": -70002,
        "ext_3": -70003,
        "ext_4": -70004,
    }
)
token = cwt.encode(
    {
        "iss": "coaps://as.example",
        "sub": "dajiaji",
```

(continues on next page)

(continued from previous page)

```
        "cti": b"123",
        "ext_1": "foo",
        "ext_2": ["bar"],
        "ext_3": {"baz": "qux"},
        "ext_4": 123,
    },
    private_key,
)
claims.set_private_claim_names(
    {
        "ext_1": -70001,
        "ext_2": -70002,
        "ext_3": -70003,
        "ext_4": -70004,
    }
)
raw = cwt.decode(token, public_key)
readable = claims.from_dict(raw)
# readable.get("ext_1") == "foo"
# readable.get("ext_2")[0] == "bar"
# readable.get("ext_3")["baz"] == "qux"
# readable.get("ext_4") == 123
```

1.2.6 CWT with PoP key

Create a CWT which has a PoP key as follows:

On the issuer side:

```
import cwt
from cwt import cose_key

# Prepares a signing key for CWT in advance.
with open("./private_key_of_issuer.pem") as key_file:
    private_key = cose_key.from_pem(key_file.read(), kid="issuer-01")

# Sets the PoP key to a CWT for the presenter.
token = cwt.encode(
    {
        "iss": "coaps://as.example",
        "sub": "dajiaji",
        "cti": "123",
        "cnf": {
            "jwk": {  # Provided by the CWT presenter.
                "kty": "OKP",
                "use": "sig",
                "crv": "Ed25519",
                "kid": "01",
                "x": "2E6dX83gqD_D0eAmqnaHe1TC1xuld6iAKXfw2OVATr0",
                "alg": "EdDSA",
            },
        },
    },
    private_key,
)
```

(continues on next page)

(continued from previous page)

```
# Issues the token to the presenter.
```

On the CWT presenter side:

```
import cwt
from cwt import cose_key

# Prepares a private PoP key in advance.
with open("./private_pop_key.pem") as key_file:
    pop_key_private = cose_key.from_pem(key_file.read(), kid="01")

# Receives a message (e.g., nonce)  from the recipient.
msg = b"could-you-sign-this-message?"  # Provided by recipient.

# Signs the message with the private PoP key.
sig = pop_key_private.sign(msg)

# Sends the msg and the sig with the CWT to the recipient.
```

On the CWT recipient side:

```
import cwt
from cwt import claims, cose_key

# Prepares the public key of the issuer in advance.
with open("./public_key_of_issuer.pem") as key_file:
    public_key = cose_key.from_pem(key_file.read(), kid="issuer-01")

# Verifies and decodes the CWT received from the presenter.
raw = cwt.decode(token, public_key)
decoded = claims.from_dict(raw)

# Extracts the PoP key from the CWT.
extracted_pop_key = cose_key.from_dict(decoded.cnf)  #  = raw[8][1]

# Then, verifies the message sent by the presenter
# with the signature which is also sent by the presenter as follows:
extracted_pop_key.verify(msg, sig)
```

In case of another PoP confirmation method `Encrypted_COSE_Key`:

```
import cwt
from cwt import claims, cose_key

with open("./private_key.pem") as key_file:
    private_key = cose_key.from_pem(key_file.read(), kid="issuer-01")

enc_key = cose_key.from_symmetric_key(
    "a-client-secret-of-cwt-recipient",  # Just 32 bytes!
    alg="ChaCha20/Poly1305",
)
pop_key = cose_key.from_symmetric_key(
    "a-client-secret-of-cwt-presenter",
    alg="HMAC 256/256",
)
```

(continues on next page)

(continued from previous page)

```
token = cwt.encode(
    {
        "iss": "coaps://as.example",
        "sub": "dajiaji",
        "cti": "123",
        "cnf": {
            # 'eck'(Encrypted Cose Key) is a keyword defined by this library.
            "eck": cose_key.to_encrypted_cose_key(pop_key, enc_key),
        },
    },
    private_key,
)

with open("./public_key.pem") as key_file:
    public_key = cose_key.from_pem(key_file.read(), kid="issuer-01")
raw = cwt.decode(token, public_key)
decoded = claims.from_dict(raw)
extracted_pop_key = cose_key.from_encrypted_cose_key(decoded.cnf, enc_key)
# extracted_pop_key.verify(message, signature)
```

In case of another PoP confirmation method `kid`:

```
import cwt
from cwt import claims, cose_key

with open("./private_key.pem") as key_file:
    private_key = cose_key.from_pem(key_file.read(), kid="issuer-01")

token = cwt.encode(
    {
        "iss": "coaps://as.example",
        "sub": "dajiaji",
        "cti": "123",
        "cnf": {
            "kid": "pop-key-id-of-cwt-presenter",
        },
    },
    private_key,
)

with open("./public_key.pem") as key_file:
    public_key = cose_key.from_pem(key_file.read(), kid="issuer-01")
raw = cwt.decode(token, public_key)
decoded = claims.from_dict(raw)
# decoded.cnf(=raw[8][3]) is kid.
```

1.3 API Reference

A Python implementation of CWT/COSE <https://python-cwt.readthedocs.io>

cwt.**encode**(*claims: Union[cwt.claims.Claims, Dict[str, Any], Dict[int, Any], bytes, str]*, *key: cwt.cose_key.COSEKey*, *nonce: bytes = b''*, *tagged: bool = False*, *recipients: Optional[List[cwt.recipient.Recipient]] = None*) → bytes

Encodes CWT with MAC, signing or encryption. This is a wrapper function of the following functions for easy use:

- *encode_and_mac*
- *encode_and_sign*
- *encode_and_encrypt*

Therefore, it must be clear whether the use of the specified key is for MAC, signing, or encryption. For this purpose, the key must have the `key_ops` parameter set to identify the usage.

Parameters

- **claims** (*Union[Claims, Dict[str, Any], Dict[int, Any], bytes, str]*) – A CWT claims object, or a JWT claims object, text string or byte string.
- **key** (*COSEKey*) – A COSE key used to generate a MAC for the claims.
- **recipients** (*List[Recipient]*) – A list of recipient information structures.
- **tagged** (*bool*) – An indicator whether the response is wrapped by CWT tag(61) or not.

Returns A byte string of the encoded CWT.

Return type bytes

Raises

- **ValueError** – Invalid arguments.
- ***EncodeError*** – Failed to encode the claims.

cwt.**encode_and_mac**(*claims: Union[cwt.claims.Claims, Dict[int, Any], bytes]*, *key: cwt.cose_key.COSEKey*, *tagged: bool = False*, *recipients: Optional[List[cwt.recipient.Recipient]] = None*) → bytes

Encodes with MAC.

Parameters

- **claims** (*Union[Claims, Dict[int, Any], bytes]*) – A CWT claims object or byte string.
- **key** (*COSEKey*) – A COSE key used to generate a MAC for the claims.
- **recipients** (*List[Recipient]*) – A list of recipient information structures.
- **tagged** (*bool*) – An indicator whether the response is wrapped by CWT tag(61) or not.

Returns A byte string of the encoded CWT.

Return type bytes

Raises

- **ValueError** – Invalid arguments.
- ***EncodeError*** – Failed to encode the claims.

cwt.**encode_and_sign**(*claims: Union[cwt.claims.Claims, Dict[int, Any], bytes]*, *key: Union[cwt.cose_key.COSEKey, List[cwt.cose_key.COSEKey]]*, *tagged: bool = False*) → bytes

Encodes CWT with signing.

Parameters

- **claims** (*Claims, Union[Dict[int, Any], bytes]*) – A CWT claims object or byte string.
- **key** (*Union[COSEKey, List[COSEKey]]*) – A COSE key or a list of the keys used to sign claims.
- **tagged** (*bool*) – An indicator whether the response is wrapped by CWT tag(61) or not.

Returns A byte string of the encoded CWT.

Return type bytes

Raises

- **ValueError** – Invalid arguments.
- ***EncodeError*** – Failed to encode the claims.

cwt.**encode_and_encrypt**(*claims: Union[cwt.claims.Claims, Dict[int, Any], bytes]*, *key: cwt.cose_key.COSEKey*, *nonce: bytes = b''*, *tagged: bool = False*, *recipients: Optional[List[cwt.recipient.Recipient]] = None*) → bytes

Encodes CWT with encryption.

Parameters

- **claims** (*Claims, Union[Dict[int, Any], bytes]*) – A CWT claims object or byte
- **string.** –
- **key** (*COSEKey*) – A COSE key used to encrypt the claims.
- **nonce** (*bytes*) – A nonce for encryption.
- **recipients** (*List[Recipient]*) – A list of recipient information structures.
- **tagged** (*bool*) – An indicator whether the response is wrapped by CWT tag(61) or not.

Returns A byte string of the encoded CWT.

Return type bytes

Raises

- **ValueError** – Invalid arguments.
- ***EncodeError*** – Failed to encode the claims.

cwt.**decode**(*data: bytes*, *key: Union[cwt.cose_key.COSEKey, List[cwt.cose_key.COSEKey]]*, *no_verify: bool = False*) → Dict[int, Any]

Verifies and decodes CWT.

Parameters

- **data** (*bytes*) – A byte string of an encoded CWT.
- **key** (*Union[COSEKey, List[COSEKey]]*) – A COSE key or a list of the keys used to verify and decrypt the encoded CWT.
- **no_verify** (*bool*) – An indicator whether token verification is skiped or not.

Returns A byte string of the decoded CWT.

Return type bytes

Raises

- **ValueError** – Invalid arguments.
- ***DecodeError*** – Failed to decode the CWT.
- ***VerifyError*** – Failed to verify the CWT.

cwt.**set_private_claim_names**(*claim_names: Dict[str, int]*)

Sets private claim definitions. This function call is redirected to the internal *ClaimsBuilder*'s *set_private_claim_names* directly. The definitions will be used in *encode* when it is called with JSON-based claims.

Parameters **claims** (*Dict[str, int]*) – A set of private claim definitions which consist of a readable claim name(str) and a claim key(int). The claim key should be less than -65536.

Raises **ValueError** – Invalid arguments.

class cwt.**CWT**(*options: Optional[Dict[str, Any]] = None*)

Bases: cwt.cbor_processor.CBORProcessor

A CWT (CBOR Web Token) Implementaion, which is built on top of *COSE*

cwt.cwt is a global object of this class initialized with default settings.

CBOR_TAG = 61

__init__(*options: Optional[Dict[str, Any]] = None*)

Constructor.

Parameters **options** (*Optional[Dict[str, Any]]*) – Options for the initial configuration of CWT. At this time, expires_in (default value: 3600) and leaway (default value: 60) are only supported. See also *expires_in*, *leeway*.

Examples

```
>>> from cwt import CWT, claims, cose_key
>>> ctx = CWT({"expires_in": 3600*24, "leeway": 10})
>>> key = cose_key.from_symmetric_key("mysecret")
>>> token = ctx.encode_and_mac(
...     claims.from_json({"iss": "coaps://as.example", "sub": "dajiaji", "cti
↪": "123"}),
...     key,
... )
```

property expires_in

The default lifetime in seconds of CWT. If *exp* is not found in claims, this value will be used with current time.

property leeway

The default leeway in seconds for validating exp and nbf.

encode(*claims: Union[cwt.claims.Claims, Dict[str, Any], Dict[int, Any], bytes, str], key: cwt.cose_key.COSEKey, nonce: bytes = b'', tagged: bool = False, recipients: Optional[List[cwt.recipient.Recipient]] = None*) → bytes

Encodes CWT with MAC, signing or encryption. This is a wrapper function of the following functions for easy use:

- *encode_and_mac*
- *encode_and_sign*
- *encode_and_encrypt*

Therefore, it must be clear whether the use of the specified key is for MAC, signing, or encryption. For this purpose, the key must have the `key_ops` parameter set to identify the usage.

Parameters

- **claims** (*Union[Claims, Dict[str, Any], Dict[int, Any], bytes, str]*) – A CWT claims object, or a JWT claims object, text string or byte string.
- **key** (*COSEKey*) – A COSE key used to generate a MAC for the claims.
- **recipients** (*List[Recipient]*) – A list of recipient information structures.
- **tagged** (*bool*) – An indicator whether the response is wrapped by CWT tag(61) or not.

Returns A byte string of the encoded CWT.

Return type bytes

Raises

- **ValueError** – Invalid arguments.
- ***EncodeError*** – Failed to encode the claims.

encode_and_mac(*claims: Union[cwt.claims.Claims, Dict[int, Any], bytes], key: cwt.cose_key.COSEKey, tagged: bool = False, recipients: Optional[List[cwt.recipient.Recipient]] = None*) → bytes

Encodes with MAC.

Parameters

- **claims** (*Union[Claims, Dict[int, Any], bytes]*) – A CWT claims object or byte string.
- **key** (*COSEKey*) – A COSE key used to generate a MAC for the claims.
- **recipients** (*List[Recipient]*) – A list of recipient information structures.
- **tagged** (*bool*) – An indicator whether the response is wrapped by CWT tag(61) or not.

Returns A byte string of the encoded CWT.

Return type bytes

Raises

- **ValueError** – Invalid arguments.
- ***EncodeError*** – Failed to encode the claims.

encode_and_sign(*claims: Union[cwt.claims.Claims, Dict[int, Any], bytes], key: Union[cwt.cose_key.COSEKey, List[cwt.cose_key.COSEKey]], tagged: bool = False*) → bytes

Encodes CWT with signing.

Parameters

- **claims** (*Claims, Union[Dict[int, Any], bytes]*) – A CWT claims object or byte string.
- **key** (*Union[COSEKey, List[COSEKey]]*) – A COSE key or a list of the keys used to sign claims.

- **tagged** (*bool*) – An indicator whether the response is wrapped by CWT tag(61) or not.

Returns A byte string of the encoded CWT.

Return type bytes

Raises

- **ValueError** – Invalid arguments.
- ***EncodeError*** – Failed to encode the claims.

encode_and_encrypt(*claims: Union[cwt.claims.Claims, Dict[int, Any], bytes]*, *key: cwt.cose_key.COSEKey*, *nonce: bytes = b''*, *tagged: bool = False*, *recipients: Optional[List[cwt.recipient.Recipient]] = None*) → bytes

Encodes CWT with encryption.

Parameters

- **claims** (*Claims*, *Union[Dict[int, Any], bytes]*) – A CWT claims object or byte
- **string.** –
- **key** (*COSEKey*) – A COSE key used to encrypt the claims.
- **nonce** (*bytes*) – A nonce for encryption.
- **recipients** (*List[Recipient]*) – A list of recipient information structures.
- **tagged** (*bool*) – An indicator whether the response is wrapped by CWT tag(61) or not.

Returns A byte string of the encoded CWT.

Return type bytes

Raises

- **ValueError** – Invalid arguments.
- ***EncodeError*** – Failed to encode the claims.

decode(*data: bytes*, *key: Union[cwt.cose_key.COSEKey, List[cwt.cose_key.COSEKey]]*, *no_verify: bool = False*) → Dict[int, Any]

Verifies and decodes CWT.

Parameters

- **data** (*bytes*) – A byte string of an encoded CWT.
- **key** (*Union[COSEKey, List[COSEKey]]*) – A COSE key or a list of the keys used to verify and decrypt the encoded CWT.
- **no_verify** (*bool*) – An indicator whether token verification is skiped or not.

Returns A byte string of the decoded CWT.

Return type bytes

Raises

- **ValueError** – Invalid arguments.
- ***DecodeError*** – Failed to decode the CWT.
- ***VerifyError*** – Failed to verify the CWT.

set_private_claim_names(*claim_names: Dict[str, int]*)
Sets private claim definitions. This function call is redirected to the internal *ClaimsBuilder*'s *set_private_claim_names* directly. The definitions will be used in *encode* when it is called with JSON-based claims.

Parameters **claims** (*Dict[str, int]*) – A set of private claim definitions which consist of a readable claim name(str) and a claim key(int). The claim key should be less than -65536.

Raises **ValueError** – Invalid arguments.

class cwt.**COSE**(*options: Optional[Dict[str, Any]] = None*)
Bases: cwt.cbor_processor.CBORProcessor

A COSE (CBOR Object Signing and Encryption) Implementaion built on top of cbor2.

cwt.cose_key is a global object of this class initialized with default settings.

__init__(*options: Optional[Dict[str, Any]] = None*)
Constructor.

At the current implementation, any options will be ignored.

encode_and_mac(*protected: Dict[int, Any], unprotected: Dict[int, Any], payload: Union[Dict[int, Any], bytes], key: cwt.cose_key.COSEKey, recipients: Optional[List[cwt.recipient.Recipient]] = None, out: str = ''*) → Union[bytes, _cbor2.CBORTag]
Encodes data with MAC.

Parameters

- **protected** (*Dict[int, Any]*) – Parameters that are to be cryptographically protected.
- **unprotected** (*Dict[int, Any]*) – Parameters that are not cryptographically protected.
- **payload** (*Union[Dict[int, Any], bytes]*) – A content to be MACed.
- **key** (COSEKey) – A COSE key as a MAC Authentication key.
- **recipients** (*Optional[List[Recipient]]*) – A list of recipient information structures.
- **out** (*str*) – An output format. Only "cbor2/CBORTag" can be used. If "cbor2/CBORTag" is specified. This function will return encoded data as cbor2's CBORTag object. If any other value is specified, it will return encoded data as bytes.

Returns A byte string of the encoded COSE or a cbor2.CBORTag object.

Return type Union[bytes, CBORTag]

Raises

- **ValueError** – Invalid arguments.
- ***EncodeError*** – Failed to encode data.

encode_and_sign(*protected: Dict[int, Any], unprotected: Dict[int, Any], payload: Union[Dict[int, Any], bytes], key: Union[cwt.cose_key.COSEKey, List[cwt.cose_key.COSEKey]], out: str = ''*) → Union[bytes, _cbor2.CBORTag]
Encodes data with signing.

Parameters

- **protected** (*Dict[int, Any]*) – Parameters that are to be cryptographically protected.

- **unprotected** (*Dict[int, Any]*) – Parameters that are not cryptographically protected.
- **payload** (*Union[Dict[int, Any], bytes]*) – A content to be signed.
- **key** (*Union[COSEKey, List[COSEKey]]*) – One or more COSE keys as signing keys.
- **out** (*str*) – An output format. Only `"cbor2/CBORTag"` can be used. If `"cbor2/CBORTag"` is specified. This function will return encoded data as cbor2's `CBORTag` object. If any other value is specified, it will return encoded data as bytes.

Returns A byte string of the encoded COSE or a cbor2.CBORTag object.

Return type Union[bytes, CBORTag]

Raises

- **ValueError** – Invalid arguments.
- ***EncodeError*** – Failed to encode data.

encode_and_encrypt (*protected: Dict[int, Any]*, *unprotected: Dict[int, Any]*, *payload: Union[Dict[int, Any], bytes]*, *key: cwt.cose_key.COSEKey*, *nonce: bytes = b''*, *recipients: Optional[List[cwt.recipient.Recipient]] = None*, *out: str = ''*) → bytes

Encodes data with encryption.

Parameters

- **protected** (*Dict[int, Any]*) – Parameters that are to be cryptographically protected.
- **unprotected** (*Dict[int, Any]*) – Parameters that are not cryptographically protected.
- **payload** (*Union[Dict[int, Any], bytes]*) – A content to be encrypted.
- **key** (*COSEKey*) – A COSE key as an encryption key.
- **nonce** (*bytes*) – A nonce for encryption.
- **recipients** (*Optional[List[Recipient]]*) – A list of recipient information structures.
- **out** (*str*) – An output format. Only `"cbor2/CBORTag"` can be used. If `"cbor2/CBORTag"` is specified. This function will return encoded data as cbor2's `CBORTag` object. If any other value is specified, it will return encoded data as bytes.

Returns A byte string of the encoded COSE or a cbor2.CBORTag object.

Return type Union[bytes, CBORTag]

Raises

- **ValueError** – Invalid arguments.
- ***EncodeError*** – Failed to encode data.

decode (*data: Union[bytes, _cbor2.CBORTag]*, *key: Union[cwt.cose_key.COSEKey, List[cwt.cose_key.COSEKey]]*) → Dict[int, Any]

Verifies and decodes COSE data.

Parameters

- **data** (*Union[bytes, CBORTag]*) – A byte string or cbor2.CBORTag of an encoded data.

- **key** (COSEKey) – A COSE key to verify and decrypt the encoded data.

Returns A decoded CBOR-like object.

Return type Dict[int, Any]

Raises

- **ValueError** – Invalid arguments.
- ***DecodeError*** – Failed to decode data.
- ***VerifyError*** – Failed to verify data.

class cwt.**Claims**(*claims: Dict[int, Any]*, *claim_names: Dict[str, int] = {'EAT-FDO': - 257, 'EAT-MAROEPrefix': - 258, 'EUPHNonce': - 259, 'aud': 3, 'cnf': 8, 'cti': 7, 'exp': 4, 'hcert': - 260, 'iat': 6, 'iss': 1, 'nbf': 5, 'sub': 2}*)

Bases: object

A class for handling CWT Claims like JWT claims.

property iss

property sub

property aud

property exp

property nbf

property iat

property cti

property cnf

get(*key: Union[str, int]*) → Any

Gets a claim value with a claim key.

Parameters **key** (*Union[str, int]*) – A claim key.

Returns The value of the claim.

Return type Any

to_dict() → Dict[int, Any]

class cwt.**ClaimsBuilder**(*options: Optional[Dict[str, Any]] = None*)

Bases: object

CBOR Web Token (CWT) Claims Builder.

cwt.claims is a global object of this class initialized with default settings.

__init__(*options: Optional[Dict[str, Any]] = None*)

Constructor.

At the current implementation, any options will be ignored.

from_dict(*claims: Dict[int, Any]*) → cwt.claims.Claims

Create a Claims object from a CBOR-like(Dict[int, Any]) claim object.

Parameters **claims** (*Dict[str, Any]*) – A CBOR-like(Dict[int, Any]) claim object.

Returns A CWT claims object.

Return type *Claims*

Raises **ValueError** – Invalid arguments.

from_json(*claims: Union[str, bytes, Dict[str, Any]]*) → cwt.claims.Claims
Converts a JWT claims object into a CWT claims object which has numeric keys. If a key string in JSON data cannot be mapped to a numeric key, it will be skipped.

Parameters **claims** (*Union[str, bytes, Dict[str, Any]]*) – A JWT claims object to be converted.

Returns A CWT claims object.

Return type *Claims*

Raises **ValueError** – Invalid arguments.

set_private_claim_names(*claim_names: Dict[str, int]*)
Sets private claim definitions. The definitions will be used in *from_json*.

Parameters **claims** (*Dict[str, int]*) – A set of private claim definitions which consist of a readable claim name(str) and a claim key(int). The claim key should be less than -65536.

Raises **ValueError** – Invalid arguments.

validate(*claims: Dict[int, Any]*)
Validates a CWT claims object.

Parameters **claims** (*Dict[int, Any]*) – A CWT claims object to be validated.

Raises **ValueError** – Failed to verify.

class cwt.**COSEKey**(*cose_key: Dict[int, Any]*)
Bases: object

The interface class for a COSE Key used for MAC, signing/verifying and encryption/decryption.

__init__(*cose_key: Dict[int, Any]*)
Constructor.

Parameters **cose_key** (*Dict[int, Any]*) – A COSE key formatted to a CBOR-like dictionary.

property key
A body of the symmetric key.

property kty
Identification of the key type.

property kid
A key identification value.

property alg
An algorithm that is used with the key.

property crv
A curve of the key type.

property key_ops
Restrict set of permissible operations.

property base_iv
Base IV to be xor-ed with Partial IVs.

to_dict() → Dict[int, Any]
Returns a CBOR-like structure (Dict[int, Any]) of the COSE key.

Returns A CBOR-like structure of the COSE key.

Return type Dict[int, Any]

generate_nonce() → bytes
Returns a nonce with a size suitable for the algorithm. This function will be called internally in *CWT* when no nonce is specified by the application. This function adopts `secrets.token_bytes()` to generate a nonce. If you do not want to use it, you should explicitly set a nonce to *CWT* functions (e.g., *encode_and_encrypt*).

Returns A byte string of a generated nonce.

Return type bytes

Raises **NotImplementedError** – Not implemented.

sign(*msg: bytes*) → bytes
Returns a digital signature for the specified message using the specified key value.

Parameters **msg** (*bytes*) – A message to be signed.

Returns A byte string of the encoded CWT.

Return type bytes

Raises

- **NotImplementedError** – Not implemented.
- **ValueError** – Invalid arguments.
- ***EncodeError*** – Failed to sign the message.

verify(*msg: bytes*, *sig: bytes*)
Verifies that the specified digital signature is valid for the specified message.

Parameters

- **msg** (*bytes*) – A message to be verified.
- **sig** (*bytes*) – A digital signature of the message.

Returns A byte string of the encoded CWT.

Return type bytes

Raises

- **NotImplementedError** – Not implemented.
- **ValueError** – Invalid arguments.
- ***VerifyError*** – Failed to verify.

encrypt(*msg: bytes*, *nonce: bytes*, *aad: bytes*) → bytes
Encrypts the specified message.

Parameters

- **msg** (*bytes*) – A message to be encrypted.
- **nonce** (*bytes*) – A nonce for encryption.
- **aad** (*bytes*) – Additional authenticated data.

Returns A byte string of encrypted data.

Return type bytes

Raises

- **NotImplementedError** – Not implemented.

- **ValueError** – Invalid arguments.
- ***EncodeError*** – Failed to encrypt the message.

decrypt(*msg: bytes*, *nonce: bytes*, *aad: bytes*) → bytes
Decrypts the specified message.

Parameters

- **msg** (*bytes*) – An encrypted message.
- **nonce** (*bytes*) – A nonce for encryption.
- **aad** (*bytes*) – Additional authenticated data.

Returns A byte string of the decrypted data.

Return type bytes

Raises

- **NotImplementedError** – Not implemented.
- **ValueError** – Invalid arguments.
- ***DecodeError*** – Failed to decrypt the message.

class cwt.**KeyBuilder**(*options: Optional[Dict[str, Any]] = None*)
Bases: cwt.cbor_processor.CBORProcessor

A *COSEKey* Builder.

__init__(*options: Optional[Dict[str, Any]] = None*)
Constructor.

At the current implementation, any `options` will be ignored.

from_symmetric_key(*key: Union[bytes, str] = b''*, *alg: Union[int, str] = 'HMAC 256/256'*, *kid: Union[bytes, str] = b''*, *key_ops: Optional[Union[List[int], List[str]]] = None*) → cwt.cose_key.COSEKey
Create a COSE key from a symmetric key.

Parameters

- **key** (*Union[bytes, str]*) – A key bytes or string.
- **alg** (*Union[int, str]*) – An algorithm label(int) or name(str). Supported `alg` are listed in Supported COSE Algorithms.
- **kid** (*Union[bytes, str]*) – A key identifier.
- **key_ops** (*Union[List[int], List[str]]*) – A list of key operation values. Following values can be used: `1("sign")`, `2("verify")`, `3("encrypt")`, `4("decrypt")`, `5("wrap key")`, `6("unwrap key")`, `7("derive key")`, `8("derive bits")`, `9("MAC create")`, `10("MAC verify")`

Returns A COSE key object.

Return type *COSEKey*

Raises **ValueError** – Invalid arguments.

from_dict(*cose_key: Dict[int, Any]*) → cwt.cose_key.COSEKey
Create a COSE key from a CBOR-like dictionary with numeric keys.

Parameters **cose_key** (*Dict[int, Any]*) – A CBOR-like dictionary with numeric keys of a COSE key.

Returns A COSE key object.

Return type *COSEKey*

Raises **ValueError** – Invalid arguments.

from_bytes(*key_data: bytes*) → cwt.cose_key.COSEKey
Create a COSE key from CBOR-formatted key data.

Parameters **key_data** (*bytes*) – CBOR-formatted key data.

Returns A COSE key object.

Return type *COSEKey*

Raises

- **ValueError** – Invalid arguments.
- ***DecodeError*** – Failed to decode the key data.

from_jwk(*data: Union[str, bytes, Dict[str, Any]]*) → cwt.cose_key.COSEKey
Create a COSE key from JWK (JSON Web Key).

Parameters **jwk** (*Union[str, bytes, Dict[str, Any]]*) – JWK-formatted key data.

Returns A COSE key object.

Return type *COSEKey*

Raises

- **ValueError** – Invalid arguments.
- ***DecodeError*** – Failed to decode the key data.

from_pem(*key_data: Union[str, bytes]*, *alg: Union[int, str] = ''*, *kid: Union[bytes, str] = b''*, *key_ops: Optional[Union[List[int], List[str]]] = None*) → cwt.cose_key.COSEKey
Create a COSE key from PEM-formatted key data.

Parameters

- **key_data** (*bytes*) – A PEM-formatted key data.
- **alg** (*Union[int, str]*) – An algorithm label(int) or name(str). Different from ::func::*cwt.KeyBuilder.from_symmetric_key*, it is only used when an algorithm cannot be specified by the PEM data, such as RSA family algorithms.
- **kid** (*Union[bytes, str]*) – A key identifier.
- **key_ops** (*Union[List[int], List[str]]*) – A list of key operation values. Following values can be used: `1("sign")`, `2("verify")`, `3("encrypt")`, `4("decrypt")`, `5("wrap key")`, `6("unwrap key")`, `7("derive key")`, `8("derive bits")`, `9("MAC create")`, `10("MAC verify")`

Returns A COSE key object.

Return type *COSEKey*

Raises

- **ValueError** – Invalid arguments.
- ***DecodeError*** – Failed to decode the key data.

from_encrypted_cose_key(*key: List[Any]*, *encryption_key: cwt.cose_key.COSEKey*) → cwt.cose_key.COSEKey

Returns an encrypted COSE key formatted to COSE_Encrypt0 structure.

Parameters

- **key** – COSEKey: A key formatted to COSE_Encrypt0 structure to be decrypted.
- **encryption_key** – COSEKey: An encryption key to decrypt the target COSE key.

Returns A COSE_Encrypt0 structure of the target COSE key.

Return type *COSEKey*

Raises

- **ValueError** – Invalid arguments.
- ***DecodeError*** – Failed to decode the COSE key.
- ***VerifyError*** – Failed to verify the COSE key.

to_encrypted_cose_key(*key: cwt.cose_key.COSEKey*, *encryption_key: cwt.cose_key.COSEKey*, *nonce: bytes = b''*, *tagged: bool = False*) → Union[List[Any], bytes]

Returns an encrypted COSE key formatted to COSE_Encrypt0 structure.

Parameters

- **key** – COSEKey: A key to be encrypted.
- **encryption_key** – COSEKey: An encryption key to encrypt the target COSE key.
- **nonce** (*bytes*) – A nonce for encryption.

Returns A COSE_Encrypt0 structure of the target COSE key.

Return type List[Any]

Raises

- **ValueError** – Invalid arguments.
- ***EncodeError*** – Failed to encrypt the COSE key.

class cwt.**Recipient**(*protected: Union[bytes, Dict[int, Any]] = {}*, *unprotected: Dict[int, Any] = {}*, *ciphertext: bytes = b''*, *recipients: List[Any] = []*)

Bases: cwt.cbor_processor.CBORProcessor

A COSE Recipient.

property protected

property unprotected

property alg

property kid

property ciphertext

property recipients

to_list() → List[Any]

exception cwt.**CWTError**

Bases: Exception

Base class for all exceptions.

exception cwt.**EncodeError**
Bases: cwt.exceptions.CWTError

An Exception occurred when a CWT/COSE encoding process failed.

exception cwt.**DecodeError**
Bases: cwt.exceptions.CWTError

An Exception occurred when a CWT/COSE decoding process failed.

exception cwt.**VerifyError**
Bases: cwt.exceptions.CWTError

An Exception occurred when a verification process failed.

1.4 Supported CWT Claims

IANA Registry for CWT Claims lists all of registered CWT claims. This section shows the claims which this library currently supports. In particular, class CWT can validate the type of the claims and Claims.from_json can convert the following Names(str) into Values(int).

1.4.1 CBOR Web Token (CWT) Claims

Name	Status	Value	Description
hcert		-260	Health Certificate
EUPHNonce		-259	Challenge Nonce defined in FIDO Device Onboarding
EATMAROEPrefix		-258	Signing prefix for multi-app restricted operating environments
EAT-FDO		-257	EAT-FDO may contain related to FIDO Device Onboarding
iss		1	Issuer
sub		2	Subject
aud		3	Audience
exp		4	Expiration Time
nbf		5	Not Before
iat		6	Issued At
cti		7	CWT ID
cnf		8	Confirmation

1.4.2 CWT Confirmation Methods

Name	Status	Value	Description
COSE_Key		1	COSE_Key Representing Public Key
Encrypted_COSE_Key		2	Encrypted COSE_Key
kid		3	Key Identifier

1.5 Supported COSE Algorithms

IANA Registry for COSE lists many cryptographic algorithms for MAC, signing, and encryption. This section shows the algorithms which this library currently supports.

1.5.1 COSE Key Types

Name	Status	Value	Description
OKP		1	Octet Key Pair
EC2		2	Elliptic Curve Keys w/ x- and y-coordinate pair
RSA		3	RSA Key
Symmetric		4	Symmetric Keys
HSS-LMS		5	Public key for HSS/LMS hash-based digital signature
WalnutDSA		6	WalnutDSA public key

1.5.2 COSE Algorithms

Name	Status	Value	Description
RS1		-65535	RSASSA-PKCS1-v1_5 using SHA-1
WalnutDSA		-260	WalnutDSA signature
RS512		-259	RSASSA-PKCS1-v1_5 using SHA-512
RS384		-258	RSASSA-PKCS1-v1_5 using SHA-384
RS256		-257	RSASSA-PKCS1-v1_5 using SHA-256
ES256K		-47	ECDSA using secp256k1 curve and SHA-256
HSS-LMS		-46	HSS/LMS hash-based digital signature
SHAKE256		-45	SHAKE-256 512-bit Hash Value
SHA-512		-44	SHA-2 512-bit Hash
SHA-384		-43	SHA-2 384-bit Hash
RSAES-OAEP w/ SHA-512		-42	RSAES-OAEP w/ SHA-512
RSAES-OAEP w/ SHA-256		-41	RSAES-OAEP w/ SHA-256

continues on next page

Table 1 – continued from previous page

Name	Status	Value	Description
RSAES-OAEP w/ RFC 8017 default parameters		-40	RSAES-OAEP w/ SHA-1
PS512		-39	RSASSA-PSS w/ SHA-512
PS384		-38	RSASSA-PSS w/ SHA-384
PS256		-37	RSASSA-PSS w/ SHA-256
ES512		-36	ECDSA w/ SHA-512
ES384		-35	ECDSA w/ SHA-384
...			
EdDSA		-8	EdDSA
ES256		-7	ECDSA w/ SHA-256
direct		-6	Direct use of CEK
...			
A128GCM		1	AES-GCM mode w/ 128-bit key, 128-bit tag
A192GCM		2	AES-GCM mode w/ 192-bit key, 128-bit tag
A256GCM		3	AES-GCM mode w/ 256-bit key, 128-bit tag
HMAC 256/64		4	HMAC w/ SHA-256 truncated to 64 bits
HMAC 256/256 ("HS256" can also be used.)		5	HMAC w/ SHA-256
HMAC 384/384 ("HS384" can also be used.)		6	HMAC w/ SHA-384
HMAC 512/512 ("HS512" can also be used.)		7	HMAC w/ SHA-512
AES-CCM-16-64-128		10	AES-CCM mode 128-bit key, 64-bit tag, 13-byte nonce

continues on next page

Table 1 – continued from previous page

Name	Status	Value	Description
AES-CCM-16-64-256		11	AES-CCM mode 256-bit key, 64-bit tag, 13-byte nonce
AES-CCM-64-64-128		12	AES-CCM mode 128-bit key, 64-bit tag, 7-byte nonce
AES-CCM-64-64-256		13	AES-CCM mode 256-bit key, 64-bit tag, 7-byte nonce
...			
ChaCha20/Poly1305		24	ChaCha20/Poly1305 w/ 256-bit key, 128-bit tag
...			
AES-CCM-16-128-128		30	AES-CCM mode 128-bit key, 128-bit tag, 13-byte nonce
AES-CCM-16-128-256		31	AES-CCM mode 256-bit key, 128-bit tag, 13-byte nonce
AES-CCM-64-128-128		32	AES-CCM mode 128-bit key, 128-bit tag, 7-byte nonce
AES-CCM-64-128-256		33	AES-CCM mode 256-bit key, 128-bit tag, 7-byte nonce

1.5.3 COSE Elliptic Curves

Name	Status	Value	Description
P-256		1	NIST P-256 also known as secp256r1
P-384		2	NIST P-384 also known as secp384r1
P-521		3	NIST P-521 also known as secp521r1
X25519		4	X25519 for use w/ ECDH only
X448		5	X448 for use w/ ECDH only
Ed25519		6	Ed25519 for use w/ EdDSA only
Ed448		7	Ed448 for use w/ EdDSA only
secp256k1		8	SECG secp256k1 curve

1.6 Referenced Specifications

This library is (partially) compliant with following specifications:

- RFC8152: CBOR Object Signing and Encryption (COSE)
- RFC8230: Using RSA Algorithms with COSE Messages
- RFC8392: CBOR Web Token (CWT)
- RFC8747: Proof-of-Possession Key Semantics for CBOR Web Tokens (CWTs)
- RFC8812: COSE and JOSE Registrations for Web Authentication (WebAuthn) Algorithms

1.7 Changes

1.7.1 Unreleased

1.7.2 Version 0.6.0

Released 2021-05-04

- Make decode accept multiple keys. #61
- Add set_private_claim_names to ClaimsBuilder and CWT. #60
- Add sample of CWT with user-defined claims to docs. #60

1.7.3 Version 0.5.0

Released 2021-05-04

- Make ClaimsBuilder return Claims. #56
- Add support for JWK keyword of alg and key_ops. #55
- Add from_jwk. #53
- Add support for PoP key (cnf claim). #50
- Add to_dict to COSEKey. #50
- Add crv property to COSEKey. #50

- Add key property to COSEKey. #50
- Add support for RSASSA-PSS. #49
- Add support for RSASSA-PKCS1-v1_5. #48

1.7.4 Version 0.4.0

Released 2021-04-30

- Add CWT.encode. #46
- Fix bug on KeyBuilder.from_dict. #45
- Add support for key_ops. #44
- Add support for ChaCha20/Poly1305. #43
- Make nonce optional for CWT.encode_and_encrypt. #42
- Add support for AES-GCM (A128GCM, A192GCM and A256GCM). #41
- Make key optional for KeyBuilder.from_symmetric_key. #41

1.7.5 Version 0.3.0

Released 2021-04-29

- Add docstring to COSE, KeyBuilder and more. #39
- Add support for COSE_Encrypt structure. #36
- Add support for COSE_Signature structure. #35
- Change protected_header type from bytes to dict. #34
- Add support for COSE_Mac structure. #32
- Add test for CWT. #29

1.7.6 Version 0.2.3

Released 2021-04-23

- Add test for cose_key and fix bugs. #21
- Add support for exp, nbf and iat. #18

1.7.7 Version 0.2.2

Released 2021-04-19

- Add support for Ed448, ES384 and ES512. #13
- Add support for EncodeError and DecodeError. #13
- Add test for supported algorithms. #13
- Update supported algorithms and claims on docs. #13

1.7.8 Version 0.2.1

Released 2021-04-18

- Add VerifyError. #11
- Fix HMAC alg names. #11
- Make COSEKey public. #11
- Add tests for HMAC. #11

1.7.9 Version 0.2.0

Released 2021-04-18

- Add docs for CWT. #9
- Raname exceptions. #9

1.7.10 Version 0.1.1

Released 2021-04-18

- Fix description of installation.

1.7.11 Version 0.1.0

Released 2021-04-18

- First public preview release.

PYTHON MODULE INDEX

C

INDEX

Symbols

A

B

C

D

E

F

G

I

K

L

M

N

P

R

S

T

U

V