Web Crypto API

> *Stability: 1 - Experimental*

Node.js provides an implementation of the standard [Web Crypto API](https://www.w3.org/TR/WebCryptoAPI/).

Use `require('crypto').webcrypto` to access this module.

```
const { subtle } = require('crypto').webcrypto;

(async function() {

  const key = await subtle.generateKey({
    name: 'HMAC',
    hash: 'SHA-256',
    length: 256
  }, true, ['sign', 'verify']);

  const enc = new TextEncoder();
  const message = enc.encode('I love cupcakes');

  const digest = await subtle.sign({
    name: 'HMAC'
  }, key, message);

})();
```

Examples

Generating keys

The {SubtleCrypto} class can be used to generate symmetric (secret) keys or asymmetric key pairs (public key and private key).

AES keys

```
const { subtle } = require('crypto').webcrypto;

async function generateAesKey(length = 256) {
  const key = await subtle.generateKey({
    name: 'AES-CBC',
    length
  }, true, ['encrypt', 'decrypt']);

  return key;
}
```

ECDSA key pairs

```
const { subtle } = require('crypto').webcrypto;

async function generateEcKey(namedCurve = 'P-521') {
  const {
    publicKey,
    privateKey
  } = await subtle.generateKey({
    name: 'ECDSA',
    namedCurve,
  }, true, ['sign', 'verify']);

  return { publicKey, privateKey };
}
```

ED25519/ED448/X25519/X448 key pairs

```
const { subtle } = require('crypto').webcrypto;

async function generateEd25519Key() {
  return subtle.generateKey({
    name: 'NODE-ED25519',
    namedCurve: 'NODE-ED25519',
  }, true, ['sign', 'verify']);
}

async function generateX25519Key() {
  return subtle.generateKey({
```

```
    name: 'ECDH',
    namedCurve: 'NODE-X25519',
  }, true, ['deriveKey']);
}
```

HMAC keys

```
const { subtle } = require('crypto').webcrypto;

async function generateHmacKey(hash = 'SHA-256') {
  const key = await subtle.generateKey({
    name: 'HMAC',
    hash
  }, true, ['sign', 'verify']);

  return key;
}
```

RSA key pairs

```
const { subtle } = require('crypto').webcrypto;
const publicExponent = new Uint8Array([1, 0, 1]);

async function generateRsaKey(modulusLength = 2048, hash = 'SHA-256') {
  const {
    publicKey,
    privateKey
  } = await subtle.generateKey({
    name: 'RSASSA-PKCS1-v1_5',
    modulusLength,
    publicExponent,
    hash,
  }, true, ['sign', 'verify']);

  return { publicKey, privateKey };
}
```

Encryption and decryption

```
const crypto = require('crypto').webcrypto;

async function aesEncrypt(plaintext) {
  const ec = new TextEncoder();
  const key = await generateAesKey();
  const iv = crypto.getRandomValues(new Uint8Array(16));

  const ciphertext = await crypto.subtle.encrypt({
    name: 'AES-CBC',
    iv,
  }, key, ec.encode(plaintext));

  return {
    key,
    iv,
    ciphertext
  };
}

async function aesDecrypt(ciphertext, key, iv) {
  const dec = new TextDecoder();
  const plaintext = await crypto.subtle.decrypt({
    name: 'AES-CBC',
    iv,
  }, key, ciphertext);

  return dec.decode(plaintext);
}
```

Exporting and importing keys

```
const { subtle } = require('crypto').webcrypto;

async function generateAndExportHmacKey(format = 'jwk', hash = 'SHA-512') {
  const key = await subtle.generateKey({
```

```
    name: 'HMAC',
    hash
  }, true, ['sign', 'verify']);

  return subtle.exportKey(format, key);
}

async function importHmacKey(keyData, format = 'jwk', hash = 'SHA-512') {
  const key = await subtle.importKey(format, keyData, {
    name: 'HMAC',
    hash
  }, true, ['sign', 'verify']);

  return key;
}
```

Wrapping and unwrapping keys

```
const { subtle } = require('crypto').webcrypto;

async function generateAndWrapHmacKey(format = 'jwk', hash = 'SHA-512') {
  const [
    key,
    wrappingKey,
  ] = await Promise.all([
    subtle.generateKey({
      name: 'HMAC', hash
    }, true, ['sign', 'verify']),
    subtle.generateKey({
      name: 'AES-KW',
      length: 256
    }, true, ['wrapKey', 'unwrapKey']),
  ]);

  const wrappedKey = await subtle.wrapKey(format, key, wrappingKey, 'AES-KW');

  return wrappedKey;
}

async function unwrapHmacKey(
  wrappedKey,
  wrappingKey,
  format = 'jwk',
  hash = 'SHA-512') {

  const key = await subtle.unwrapKey(
    format,
    wrappedKey,
    unwrappingKey,
    'AES-KW',
    { name: 'HMAC', hash },
    true,
    ['sign', 'verify']);

  return key;
}
```

Sign and verify

```
const { subtle } = require('crypto').webcrypto;

async function sign(key, data) {
  const ec = new TextEncoder();
  const signature =
    await subtle.sign('RSASSA-PKCS1-v1_5', key, ec.encode(data));
  return signature;
}

async function verify(key, signature, data) {
  const ec = new TextEncoder();
  const verified =
    await subtle.verify(
      'RSASSA-PKCS1-v1_5',
      key,
      signature,
      ec.encode(data));
```

```
  return verified;
}
```

Deriving bits and keys

```
const { subtle } = require('crypto').webcrypto;

async function pbkdf2(pass, salt, iterations = 1000, length = 256) {
  const ec = new TextEncoder();
  const key = await subtle.importKey(
    'raw',
    ec.encode(pass),
    'PBKDF2',
    false,
    ['deriveBits']);
  const bits = await subtle.deriveBits({
    name: 'PBKDF2',
    hash: 'SHA-512',
    salt: ec.encode(salt),
    iterations
  }, key, length);
  return bits;
}

async function pbkdf2Key(pass, salt, iterations = 1000, length = 256) {
  const ec = new TextEncoder();
  const keyMaterial = await subtle.importKey(
    'raw',
    ec.encode(pass),
    'PBKDF2',
    false,
    ['deriveKey']);
  const key = await subtle.deriveKey({
    name: 'PBKDF2',
    hash: 'SHA-512',
    salt: ec.encode(salt),
    iterations
  }, keyMaterial, {
    name: 'AES-GCM',
    length: 256
  }, true, ['encrypt', 'decrypt']);
  return key;
}
```

Digest

```
const { subtle } = require('crypto').webcrypto;

async function digest(data, algorithm = 'SHA-512') {
  const ec = new TextEncoder();
  const digest = await subtle.digest(algorithm, ec.encode(data));
  return digest;
}
```

Algorithm matrix

The table details the algorithms supported by the Node.js Web Crypto API implementation and the APIs supported for each:

Algorithm	`generateKey`	`exportKey`	`importKey`	`encrypt`	`decrypt`	`wrapKey`	`unwrapKey`	`deriveBits`	`deriveKey`	`sign`	`verify`	`dige`
`'RSASSA-PKCS1-v1_5'`	✓	✓	✓							✓	✓	
`'RSA-PSS'`	✓	✓	✓							✓	✓	
`'RSA-OAEP'`	✓	✓	✓	✓	✓	✓	✓					
`'ECDSA'`	✓	✓	✓							✓	✓	
`'ECDH'`	✓	✓	✓					✓	✓			
`'AES-CTR'`	✓	✓	✓	✓	✓	✓	✓					
`'AES-CBC'`	✓	✓	✓	✓	✓	✓	✓					
`'AES-GCM'`	✓	✓	✓	✓	✓	✓	✓					
`'AES-KW'`	✓	✓	✓			✓	✓					

`'HMAC'`	✓	✓	✓							✓	✓	
`'HKDF'`		✓	✓					✓	✓			
`'PBKDF2'`		✓	✓					✓	✓			
`'SHA-1'`												✓
`'SHA-256'`												✓
`'SHA-384'`												✓
`'SHA-512'`												✓
`'NODE-DSA'`[^1]	✓	✓	✓							✓	✓	
`'NODE-DH'`[^1]	✓	✓	✓					✓	✓			
`'NODE-ED25519'`[^1]	✓	✓	✓							✓	✓	
`'NODE-ED448'`[^1]	✓	✓	✓							✓	✓	

Class: `Crypto`

Calling `require('crypto').webcrypto` returns an instance of the `Crypto` class. `Crypto` is a singleton that provides access to the remainder of the crypto API.

`crypto.subtle`

- Type: {SubtleCrypto}

Provides access to the `SubtleCrypto` API.

`crypto.getRandomValues(typedArray)`

- `typedArray` {Buffer|TypedArray}
- Returns: {Buffer|TypedArray}

Generates cryptographically strong random values. The given `typedArray` is filled with random values, and a reference to `typedArray` is returned.

The given `typedArray` must be an integer-based instance of {TypedArray}, i.e. `Float32Array` and `Float64Array` are not accepted.

An error will be thrown if the given `typedArray` is larger than 65,536 bytes.

`crypto.randomUUID()`

- Returns: {string}

Generates a random RFC 4122 version 4 UUID. The UUID is generated using a cryptographic pseudorandom number generator.

Class: `CryptoKey`

`cryptoKey.algorithm`

- Type: {AesKeyGenParams|RsaHashedKeyGenParams|EcKeyGenParams|HmacKeyGenParams|NodeDsaKeyGenParams|NodeDhKeyGenParams}

An object detailing the algorithm for which the key can be used along with additional algorithm-specific parameters.

Read-only.

`cryptoKey.extractable`

- Type: {boolean}

When `true`, the {CryptoKey} can be extracted using either `subtleCrypto.exportKey()` or `subtleCrypto.wrapKey()`.

Read-only.

`cryptoKey.type`

- Type: {string} One of `'secret'`, `'private'`, or `'public'`.

A string identifying whether the key is a symmetric (`'secret'`) or asymmetric (`'private'` or `'public'`) key.

`cryptoKey.usages`

- Type: {string[]}

An array of strings identifying the operations for which the key may be used.

The possible usages are:

- `'encrypt'` - The key may be used to encrypt data.
- `'decrypt'` - The key may be used to decrypt data.

* `'sign'` - The key may be used to generate digital signatures.
* `'verify'` - The key may be used to verify digital signatures.
* `'deriveKey'` - The key may be used to derive a new key.
* `'deriveBits'` - The key may be used to derive bits.
* `'wrapKey'` - The key may be used to wrap another key.
* `'unwrapKey'` - The key may be used to unwrap another key.

Valid key usages depend on the key algorithm (identified by `cryptokey.algorithm.name`).

Key Type	`'encrypt'`	`'decrypt'`	`'sign'`	`'verify'`	`'deriveKey'`	`'deriveBits'`	`'wrapKey'`	`'unwrapKey'`
`'AES-CBC'`	✓	✓					✓	✓
`'AES-CTR'`	✓	✓					✓	✓
`'AES-GCM'`	✓	✓					✓	✓
`'AES-KW'`							✓	✓
`'ECDH'`					✓	✓		
`'ECDSA'`			✓	✓				
`'HDKF'`					✓	✓		
`'HMAC'`			✓	✓				
`'PBKDF2'`					✓	✓		
`'RSA-OAEP'`	✓	✓					✓	✓
`'RSA-PSS'`			✓	✓				
`'RSASSA-PKCS1-v1_5'`			✓	✓				
`'NODE-DSA'`[^1]			✓	✓				
`'NODE-DH'`[^1]					✓	✓		
`'NODE-SCRYPT'`[^1]					✓	✓		
`'NODE-ED25519'`[^1]			✓	✓				
`'NODE-ED448'`[^1]			✓	✓				

Class: `CryptoKeyPair`

The `CryptoKeyPair` is a simple dictionary object with `publicKey` and `privateKey` properties, representing an asymmetric key pair.

`cryptoKeyPair.privateKey`

* Type: {CryptoKey} A {CryptoKey} whose `type` will be `'private'`.

`cryptoKeyPair.publicKey`

* Type: {CryptoKey} A {CryptoKey} whose `type` will be `'public'`.

Class: `SubtleCrypto`

`subtle.decrypt(algorithm, key, data)`

* `algorithm`: {RsaOaepParams|AesCtrParams|AesCbcParams|AesGcmParams}
* `key`: {CryptoKey}
* `data`: {ArrayBuffer|TypedArray|DataView|Buffer}
* Returns: {Promise} containing {ArrayBuffer}

Using the method and parameters specified in `algorithm` and the keying material provided by `key`, `subtle.decrypt()` attempts to decipher the provided `data`. If successful, the returned promise will be resolved with an {ArrayBuffer} containing the plaintext result.

The algorithms currently supported include:

* `'RSA-OAEP'`
* `'AES-CTR'`
* `'AES-CBC'`
* `'AES-GCM`'

`subtle.deriveBits(algorithm, baseKey, length)`

* `algorithm`: {EcdhKeyDeriveParams|HkdfParams|Pbkdf2Params|NodeDhDeriveBitsParams|NodeScryptParams}
* `baseKey`: {CryptoKey}
* `length`: {number}
* Returns: {Promise} containing {ArrayBuffer}

Using the method and parameters specified in `algorithm` and the keying material provided by `baseKey`, `subtle.deriveBits()` attempts to generate `length` bits. The Node.js implementation requires that `length` is a multiple of `8`. If successful, the returned promise will be resolved with an {ArrayBuffer} containing the generated data.

The algorithms currently supported include:

- `'ECDH'`
- `'HKDF'`
- `'PBKDF2'`
- `'NODE-DH'` [^1]
- `'NODE-SCRYPT'` [^1]

`subtle.deriveKey(algorithm, baseKey, derivedKeyAlgorithm, extractable, keyUsages)`

- `algorithm` : {EcdhKeyDeriveParams|HkdfParams|Pbkdf2Params|NodeDhDeriveBitsParams|NodeScryptParams}
- `baseKey` : {CryptoKey}
- `derivedKeyAlgorithm` : {HmacKeyGenParams|AesKeyGenParams}
- `extractable` : {boolean}
- `keyUsages` : {string[]} See [Key usages](#key-usages).
- Returns: {Promise} containing {CryptoKey}

Using the method and parameters specified in `algorithm` , and the keying material provided by `baseKey` , `subtle.deriveKey()` attempts to generate a new {CryptoKey} based on the method and parameters in `derivedKeyAlgorithm` .

Calling `subtle.deriveKey()` is equivalent to calling `subtle.deriveBits()` to generate raw keying material, then passing the result into the `subtle.importKey()` method using the `deriveKeyAlgorithm` , `extractable` , and `keyUsages` parameters as input.

The algorithms currently supported include:

- `'ECDH'`
- `'HKDF'`
- `'PBKDF2'`
- `'NODE-DH'` [^1]
- ' `NODE-SCRYPT'` [^1]

`subtle.digest(algorithm, data)`

- `algorithm` : {string|Object}
- `data` : {ArrayBuffer|TypedArray|DataView|Buffer}
- Returns: {Promise} containing {ArrayBuffer}

Using the method identified by `algorithm` , `subtle.digest()` attempts to generate a digest of `data` . If successful, the returned promise is resolved with an {ArrayBuffer} containing the computed digest.

If `algorithm` is provided as a {string}, it must be one of:

- `'SHA-1'`
- `'SHA-256'`
- `'SHA-384'`
- `'SHA-512'`

If `algorithm` is provided as an {Object}, it must have a `name` property whose value is one of the above.

`subtle.encrypt(algorithm, key, data)`

- `algorithm` : {RsaOaepParams|AesCtrParams|AesCbcParams|AesGcmParams}
- `key` : {CryptoKey}
- Returns: {Promise} containing {ArrayBuffer}

Using the method and parameters specified by `algorithm` and the keying material provided by `key` , `subtle.encrypt()` attempts to encipher `data` . If successful, the returned promise is resolved with an {ArrayBuffer} containing the encrypted result.

The algorithms currently supported include:

- `'RSA-OAEP'`
- `'AES-CTR'`
- `'AES-CBC'`
- `'AES-GCM` '

`subtle.exportKey(format, key)`

- `format` : {string} Must be one of `'raw'` , `'pkcs8'` , `'spki'` , `'jwk'` , or `'node.keyObject'` .
- `key` : {CryptoKey}
- Returns: {Promise} containing {ArrayBuffer}, or, if `format` is `'node.keyObject'` , a {KeyObject}.

Exports the given key into the specified format, if supported.

If the {CryptoKey} is not extractable, the returned promise will reject.

When `format` is either `'pkcs8'` or `'spki'` and the export is successful, the returned promise will be resolved with an {ArrayBuffer} containing the exported key data.

When `format` is `'jwk'` and the export is successful, the returned promise will be resolved with a JavaScript object conforming to the [JSON Web Key](https://tools.ietf.org/html/rfc7517) specification.

The special `'node.keyObject'` value for `format` is a Node.js-specific extension that allows converting a {CryptoKey} into a Node.js {KeyObject}.

Key Type	`'spki'`	`'pkcs8'`	`'jwk'`	`'raw'`
`'AES-CBC'`			✓	✓
`'AES-CTR'`			✓	✓
`'AES-GCM'`			✓	✓
`'AES-KW'`			✓	✓
`'ECDH'`	✓	✓	✓	✓
`'ECDSA'`	✓	✓	✓	✓
`'HDKF'`				
`'HMAC'`			✓	✓
`'PBKDF2'`				
`'RSA-OAEP'`	✓	✓	✓	
`'RSA-PSS'`	✓	✓	✓	
`'RSASSA-PKCS1-v1_5'`	✓	✓	✓	
`'NODE-DSA'`[^1]	✓	✓		
`'NODE-DH'`[^1]	✓	✓		
`'NODE-SCRYPT'`[^1]				
`'NODE-ED25519'`[^1]	✓	✓	✓	✓
`'NODE-ED448'`[^1]	✓	✓	✓	✓

`subtle.generateKey(algorithm, extractable, keyUsages)`

- `algorithm` : {RsaHashedKeyGenParams|EcKeyGenParams|HmacKeyGenParams|AesKeyGenParams|NodeDsaKeyGenParams|NodeDhKeyGenParams|NodeEdKeyGenParams}
- `extractable` : {boolean}
- `keyUsages` : {string[]} See Key usages.
- Returns: {Promise} containing {CryptoKey|CryptoKeyPair}

Using the method and parameters provided in `algorithm`, `subtle.generateKey()` attempts to generate new keying material. Depending the method used, the method may generate either a single {CryptoKey} or a {CryptoKeyPair}.

The {CryptoKeyPair} (public and private key) generating algorithms supported include:

- `'RSASSA-PKCS1-v1_5'`
- `'RSA-PSS'`
- `'RSA-OAEP'`
- `'ECDSA'`
- `'ECDH'`
- `'NODE-DSA'` [^1]
- `'NODE-DH'` [^1]
- `'NODE-ED25519'` [^1]
- `'NODE-ED448'` [^1]

The {CryptoKey} (secret key) generating algorithms supported include:

- `'HMAC'`
- `'AES-CTR'`
- `'AES-CBC'`
- `'AES-GCM'`
- `'AES-KW'`

`subtle.importKey(format, keyData, algorithm, extractable, keyUsages)`

- `format` : {string} Must be one of `'raw'`, `'pkcs8'`, `'spki'`, `'jwk'`, or `'node.keyObject'`.
- `keyData` : {ArrayBuffer|TypedArray|DataView|Buffer|KeyObject}
- `algorithm` : {RsaHashedImportParams|EcKeyImportParams|HmacImportParams|AesImportParams|Pbkdf2ImportParams|NodeDsaImportParams|NodeDhImportParams|NodeScryptImpo
- `extractable` : {boolean}
- `keyUsages` : {string[]} See Key usages.
- Returns: {Promise} containing {CryptoKey}

The `subtle.importKey()` method attempts to interpret the provided `keyData` as the given `format` to create a {CryptoKey} instance using the provided `algorithm`, `extractable`, and `keyUsages` arguments. If the import is successful, the returned promise will be resolved with the created {CryptoKey}.

The special `'node.keyObject'` value for `format` is a Node.js-specific extension that allows converting a Node.js {KeyObject} into a {CryptoKey}.

If importing a `'PBKDF2'` key, `extractable` must be `false`.

The algorithms currently supported include:

Key Type	`'spki'`	`'pkcs8'`	`'jwk'`	`'raw'`
`'AES-CBC'`			✓	✓
`'AES-CTR'`			✓	✓
`'AES-GCM'`			✓	✓
`'AES-KW'`			✓	✓
`'ECDH'`	✓	✓	✓	✓
`'ECDSA'`	✓	✓	✓	✓
`'HDKF'`				✓
`'HMAC'`			✓	✓
`'PBKDF2'`				✓
`'RSA-OAEP'`	✓	✓	✓	
`'RSA-PSS'`	✓	✓	✓	
`'RSASSA-PKCS1-v1_5'`	✓	✓	✓	
`'NODE-DSA'`[^1]	✓	✓		
`'NODE-DH'`[^1]	✓	✓		
`'NODE-SCRYPT'`[^1]				✓
`'NODE-ED25519'`[^1]	✓	✓	✓	✓
`'NODE-ED448'`[^1]	✓	✓	✓	✓

`subtle.sign(algorithm, key, data)`

- `algorithm` : {RsaSignParams|RsaPssParams|EcdsaParams|HmacParams|NodeDsaSignParams}
- `key` : {CryptoKey}
- `data` : {ArrayBuffer|TypedArray|DataView|Buffer}
- Returns: {Promise} containing {ArrayBuffer}

Using the method and parameters given by `algorithm` and the keying material provided by `key`, `subtle.sign()` attempts to generate a cryptographic signature of `data`. If successful, the returned promise is resolved with an {ArrayBuffer} containing the generated signature.

The algorithms currently supported include:

- `'RSASSA-PKCS1-v1_5'`
- `'RSA-PSS'`
- `'ECDSA'`
- `'HMAC'`
- `'NODE-DSA'` [^1]
- `'NODE-ED25519'` [^1]
- `'NODE-ED448'` [^1]

`subtle.unwrapKey(format, wrappedKey, unwrappingKey, unwrapAlgo, unwrappedKeyAlgo, extractable, keyUsages)`

- `format` : {string} Must be one of `'raw'`, `'pkcs8'`, `'spki'`, or `'jwk'`.
- `wrappedKey` : {ArrayBuffer|TypedArray|DataView|Buffer}
- `unwrappingKey` : {CryptoKey}
- `unwrapAlgo` : {RsaOaepParams|AesCtrParams|AesCbcParams|AesGcmParams|AesKwParams}
- `unwrappedKeyAlgo` : {RsaHashedImportParams|EcKeyImportParams|HmacImportParams|AesImportParams}
- `extractable` : {boolean}
- `keyUsages` : {string[]} See Key usages.
- Returns: {Promise} containing {CryptoKey}

In cryptography, "wrapping a key" refers to exporting and then encrypting the keying material. The `subtle.unwrapKey()` method attempts to decrypt a wrapped key and create a {CryptoKey} instance. It is equivalent to calling `subtle.decrypt()` first on the encrypted key data (using the `wrappedKey`, `unwrapAlgo`, and `unwrappingKey` arguments as input) then passing the results in to the `subtle.importKey()` method using the `unwrappedKeyAlgo`, `extractable`, and `keyUsages` arguments as inputs. If successful, the returned promise is resolved with a {CryptoKey} object.

The wrapping algorithms currently supported include:

- `'RSA-OAEP'`
- `'AES-CTR'`
- `'AES-CBC'`
- `'AES-GCM'`
- `'AES-KW'`

The unwrapped key algorithms supported include:

- `'RSASSA-PKCS1-v1_5'`
- `'RSA-PSS'`
- `'RSA-OAEP'`
- `'ECDSA'`
- `'ECDH'`
- `'HMAC'`
- `'AES-CTR'`
- `'AES-CBC'`
- `'AES-GCM'`
- `'AES-KW'`
- `'NODE-DSA'` [^1]
- `'NODE-DH'` [^1]

`subtle.verify(algorithm, key, signature, data)`

- `algorithm` : {RsaSignParams|RsaPssParams|EcdsaParams|HmacParams|NodeDsaSignParams}
- `key` : {CryptoKey}
- `signature` : {ArrayBuffer|TypedArray|DataView|Buffer}
- `data` : {ArrayBuffer|TypedArray|DataView|Buffer}
- Returns: {Promise} containing {boolean}

Using the method and parameters given in `algorithm` and the keying material provided by `key` , `subtle.verify()` attempts to verify that `signature` is a valid cryptographic signature of `data` . The returned promise is resolved with either `true` or `false` .

The algorithms currently supported include:

- `'RSASSA-PKCS1-v1_5'`
- `'RSA-PSS'`
- `'ECDSA'`
- `'HMAC'`
- `'NODE-DSA'` [^1]
- `'NODE-ED25519'` [^1]
- `'NODE-ED448'` [^1]

`subtle.wrapKey(format, key, wrappingKey, wrapAlgo)`

- `format` : {string} Must be one of `'raw'` , `'pkcs8'` , `'spki'` , or `'jwk'` .
- `key` : {CryptoKey}
- `wrappingKey` : {CryptoKey}
- `wrapAlgo` : {RsaOaepParams|AesCtrParams|AesCbcParams|AesGcmParams|AesKwParams}
- Returns: {Promise} containing {ArrayBuffer}

In cryptography, "wrapping a key" refers to exporting and then encrypting the keying material. The `subtle.wrapKey()` method exports the keying material into the format identified by `format` , then encrypts it using the method and parameters specified by `wrapAlgo` and the keying material provided by `wrappingKey` . It is the equivalent to calling `subtle.exportKey()` using `format` and `key` as the arguments, then passing the result to the `subtle.encrypt()` method using `wrappingKey` and `wrapAlgo` as inputs. If successful, the returned promise will be resolved with an {ArrayBuffer} containing the encrypted key data.

The wrapping algorithms currently supported include:

- `'RSA-OAEP'`
- `'AES-CTR'`
- `'AES-CBC'`
- `'AES-GCM'`
- `'AES-KW'`

Algorithm parameters

The algorithm parameter objects define the methods and parameters used by the various {SubtleCrypto} methods. While described here as "classes", they are simple JavaScript dictionary objects.

Class: `AesCbcParams`

`aesCbcParams.iv`

- Type: {ArrayBuffer|TypedArray|DataView|Buffer}

Provides the initialization vector. It must be exactly 16-bytes in length and should be unpredictable and cryptographically random.

`aesCbcParams.name`

- Type: {string} Must be `'AES-CBC'` .

Class: `AesCtrParams`

`aesCtrParams.counter`

- Type: {ArrayBuffer|TypedArray|DataView|Buffer}

The initial value of the counter block. This must be exactly 16 bytes long.

The `AES-CTR` method uses the rightmost `length` bits of the block as the counter and the remaining bits as the nonce.

`aesCtrParams.length`

- Type: {number} The number of bits in the `aesCtrParams.counter` that are to be used as the counter.

`aesCtrParams.name`

- Type: {string} Must be `'AES-CTR'` .

Class: `AesGcmParams`

`aesGcmParams.additionalData`

- Type: {ArrayBuffer|TypedArray|DataView|Buffer|undefined}

With the AES-GCM method, the `additionalData` is extra input that is not encrypted but is included in the authentication of the data. The use of `additionalData` is optional.

`aesGcmParams.iv`

- Type: {ArrayBuffer|TypedArray|DataView|Buffer}

The initialization vector must be unique for every encryption operation using a given key. The AES-GCM specification recommends that this contain at least 12 random bytes.

`aesGcmParams.name`

- Type: {string} Must be `'AES-GCM'` .

`aesGcmParams.tagLength`

- Type: {number} The size in bits of the generated authentication tag. This values must be one of `32` , `64` , `96` , `104` , `112` , `120` , or `128` . **Default:** `128` .

Class: `AesImportParams`

`aesImportParams.name`

- Type: {string} Must be one of `'AES-CTR'` , `'AES-CBC'` , `'AES-GCM'` , or `'AES-KW'` .

Class: `AesKeyGenParams`

`aesKeyGenParams.length`

- Type: {number}

The length of the AES key to be generated. This must be either `128` , `192` , or `256` .

`aesKeyGenParams.name`

- Type: {string} Must be one of `'AES-CBC'` , `'AES-CTR'` , `'AES-GCM'` , or `'AES-KW'`

Class: `AesKwParams`

`aesKwParams.name`

- Type: {string} Must be `'AES-KW'` .

Class: `EcdhKeyDeriveParams`

`ecdhKeyDeriveParams.name`

- Type: {string} Must be `'ECDH'` .

`ecdhKeyDeriveParams.public`

- Type: {CryptoKey}

ECDH key derivation operates by taking as input one parties private key and another parties public key -- using both to generate a common shared secret. The `ecdhKeyDeriveParams.public` property is set to the other parties public key.

Class: `EcdsaParams`

`ecdsaParams.hash`

- Type: {string|Object}

If represented as a {string}, the value must be one of:

- `'SHA-1'`
- `'SHA-256'`
- `'SHA-384'`
- `'SHA-512'`

If represented as an {Object}, the object must have a `name` property whose value is one of the above listed values.

`ecdsaParams.name`

- Type: {string} Must be `'ECDSA'` .

Class: `EcKeyGenParams`

`ecKeyGenParams.name`

- Type: {string} Must be one of `'ECDSA'` or `'ECDH'` .

`ecKeyGenParams.namedCurve`

- Type: {string} Must be one of `'P-256'`, `'P-384'`, `'P-521'`, `'NODE-ED25519'`, `'NODE-ED448'`, `'NODE-X25519'`, or `'NODE-X448'`.

Class: `EcKeyImportParams`

`ecKeyImportParams.name`

- Type: {string} Must be one of `'ECDSA'` or `'ECDH'`.

`ecKeyImportParams.namedCurve`

- Type: {string} Must be one of `'P-256'`, `'P-384'`, `'P-521'`, `'NODE-ED25519'`, `'NODE-ED448'`, `'NODE-X25519'`, or `'NODE-X448'`.

Class: `HkdfParams`

`hkdfParams.hash`

- Type: {string|Object}

If represented as a {string}, the value must be one of:

- `'SHA-1'`
- `'SHA-256'`
- `'SHA-384'`
- `'SHA-512'`

If represented as an {Object}, the object must have a `name` property whose value is one of the above listed values.

`hkdfParams.info`

- Type: {ArrayBuffer|TypedArray|DataView|Buffer}

Provides application-specific contextual input to the HKDF algorithm. This can be zero-length but must be provided.

`hkdfParams.name`

- Type: {string} Must be `'HKDF'`.

`hkdfParams.salt`

- Type: {ArrayBuffer|TypedArray|DataView|Buffer}

The salt value significantly improves the strength of the HKDF algorithm. It should be random or pseudorandom and should be the same length as the output of the digest function (for instance, if using `'SHA-256'` as the digest, the salt should be 256-bits of random data).

Class: `HmacImportParams`

`hmacImportParams.hash`

- Type: {string|Object}

If represented as a {string}, the value must be one of:

- `'SHA-1'`
- `'SHA-256'`
- `'SHA-384'`
- `'SHA-512'`

If represented as an {Object}, the object must have a `name` property whose value is one of the above listed values.

`hmacImportParams.length`

- Type: {number}

The optional number of bits in the HMAC key. This is optional and should be omitted for most cases.

`hmacImportParams.name`

- Type: {string} Must be `'HMAC'`.

Class: `HmacKeyGenParams`

`hmacKeyGenParams.hash`

- Type: {string|Object}

If represented as a {string}, the value must be one of:

- `'SHA-1'`
- `'SHA-256'`
- `'SHA-384'`
- `'SHA-512'`

If represented as an {Object}, the object must have a `name` property whose value is one of the above listed values.

`hmacKeyGenParams.length`

- Type: {number}

The number of bits to generate for the HMAC key. If omitted, the length will be determined by the hash algorithm used. This is optional and should be omitted for most cases.

`hmacKeyGenParams.name`

- Type: {string} Must be `'HMAC'`.

Class: `HmacParams`

`hmacParams.name`

- Type: {string} Must be `'HMAC'` .

Class: `Pbkdf2ImportParams`

`pbkdf2ImportParams.name`

- Type: {string} Must be `'PBKDF2'`

Class: `Pbkdf2Params`

`pbkdb2Params.hash`

- Type: {string|Object}

If represented as a {string}, the value must be one of:

- `'SHA-1'`
- `'SHA-256'`
- `'SHA-384'`
- `'SHA-512'`

If represented as an {Object}, the object must have a `name` property whose value is one of the above listed values.

`pbkdf2Params.iterations`

- Type: {number}

The number of iterations the PBKDF2 algorithm should make when deriving bits.

`pbkdf2Params.name`

- Type: {string} Must be `'PBKDF2'` .

`pbkdf2Params.salt`

- Type: {ArrayBuffer|TypedArray|DataView|Buffer}

Should be at least 16 random or pseudorandom bytes.

Class: `RsaHashedImportParams`

`rsaHashedImportParams.hash`

- Type: {string|Object}

If represented as a {string}, the value must be one of:

- `'SHA-1'`
- `'SHA-256'`
- `'SHA-384'`
- `'SHA-512'`

If represented as an {Object}, the object must have a `name` property whose value is one of the above listed values.

`rsaHashedImportParams.name`

- Type: {string} Must be one of `'RSASSA-PKCS1-v1_5'` , `'RSA-PSS'` , or `'RSA-OAEP'` .

Class: `RsaHashedKeyGenParams`

`rsaHashedKeyGenParams.hash`

- Type: {string|Object}

If represented as a {string}, the value must be one of:

- `'SHA-1'`
- `'SHA-256'`
- `'SHA-384'`
- `'SHA-512'`

If represented as an {Object}, the object must have a `name` property whose value is one of the above listed values.

`rsaHashedKeyGenParams.modulusLength`

- Type: {number}

The length in bits of the RSA modulus. As a best practice, this should be at least `2048` .

`rsaHashedKeyGenParams.name`

- Type: {string} Must be one of `'RSASSA-PKCS1-v1_5'` , `'RSA-PSS'` , or `'RSA-OAEP'` .

`rsaHashedKeyGenParams.publicExponent`

- Type: {Uint8Array}

The RSA public exponent. This must be a {Uint8Array} containing a big-endian, unsigned integer that must fit within 32-bits. The {Uint8Array} may contain an arbitrary number of leading zero-bits. The value must be a prime number. Unless there is reason to use a different value, use `new Uint8Array([1, 0, 1])` (65537) as the public exponent.

Class: `RsaOaepParams`

rsaOaepParams.label

- Type: {ArrayBuffer|TypedArray|DataView|Buffer}

An additional collection of bytes that will not be encrypted, but will be bound to the generated ciphertext.

The `rsaOaepParams.label` parameter is optional.

rsaOaepParams.name

- Type: {string} must be `'RSA-OAEP'` .

Class: `RsaPssParams`

`rsaPssParams.name`

- Type: {string} Must be `'RSA-PSS'` .

`rsaPssParams.saltLength`

- Type: {number}

The length (in bytes) of the random salt to use.

Class: `RsaSignParams`

`rsaSignParams.name`

- Type: {string} Must be `'RSASSA-PKCS1-v1_5'`

Node.js-specific extensions

The Node.js Web Crypto API extends various aspects of the Web Crypto API. These extensions are consistently identified by prepending names with the `node.` prefix. For instance, the `'node.keyObject'` key format can be used with the `subtle.exportKey()` and `subtle.importKey()` methods to convert between a WebCrypto {CryptoKey} object and a Node.js {KeyObject}.

Care should be taken when using Node.js-specific extensions as they are not supported by other WebCrypto implementations and reduce the portability of code to other environments.

`NODE-DH` Algorithm

The `NODE-DH` algorithm is the common implementation of Diffie-Hellman key agreement.

Class: `NodeDhImportParams`

`nodeDhImportParams.name`

- Type: {string} Must be `'NODE-DH'` .

Class: `NodeDhKeyGenParams`

`nodeDhKeyGenParams.generator`

- Type: {number} A custom generator.

`nodeDhKeyGenParams.group`

- Type: {string} The Diffie-Hellman group name.

`nodeDhKeyGenParams.prime`

- Type: {Buffer} The prime parameter.

`nodeDhKeyGenParams.primeLength`

- Type: {number} The length in bits of the prime.

Class: NodeDhDeriveBitsParams

`nodeDhDeriveBitsParams.public`

- Type: {CryptoKey} The other parties public key.

`NODE-DSA` Algorithm

The `NODE-DSA` algorithm is the common implementation of the DSA digital signature algorithm.

Class: `NodeDsaImportParams`

`nodeDsaImportParams.hash`

- Type: {string|Object}

If represented as a {string}, the value must be one of:

- `'SHA-1'`
- `'SHA-256'`
- `'SHA-384'`
- `'SHA-512'`

If represented as an {Object}, the object must have a `name` property whose value is one of the above listed values.

`nodeDsaImportParams.name`

* Type: {string} Must be `'NODE-DSA'`.

Class: `NodeDsaKeyGenParams`

`nodeDsaKeyGenParams.divisorLength`

* Type: {number}

The optional length in bits of the DSA divisor.

`nodeDsaKeyGenParams.hash`

* Type: {string|Object}

If represented as a {string}, the value must be one of:

* `'SHA-1'`
* `'SHA-256'`
* `'SHA-384'`
* `'SHA-512'`

If represented as an {Object}, the object must have a `name` property whose value is one of the above listed values.

`nodeDsaKeyGenParams.modulusLength`

* Type: {number}

The length in bits of the DSA modulus. As a best practice, this should be at least `2048`.

`nodeDsaKeyGenParams.name`

* Type: {string} Must be `'NODE-DSA'`.

Class: `NodeDsaSignParams`

`nodeDsaSignParams.name`

* Type: {string} Must be `'NODE-DSA'`

`NODE-ED25519` and `NODE-ED448` Algorithms

Class: `NodeEdKeyGenParams`

`nodeEdKeyGenParams.name`

* Type: {string} Must be one of `'NODE-ED25519'`, `'NODE-ED448'` or `'ECDH'`.

`nodeEdKeyGenParams.namedCurve`

* Type: {string} Must be one of `'NODE-ED25519'`, `'NODE-ED448'`, `'NODE-X25519'`, or `'NODE-X448'`.

Class: `NodeEdKeyImportParams`

`nodeEdKeyImportParams.name`

* Type: {string} Must be one of `'NODE-ED25519'` or `'NODE-ED448'` if importing an `Ed25519` or `Ed448` key, or `'ECDH'` if importing an `X25519` or `X448` key.

`nodeEdKeyImportParams.namedCurve`

* Type: {string} Must be one of `'NODE-ED25519'`, `'NODE-ED448'`, `'NODE-X25519'`, or `'NODE-X448'`.

`nodeEdKeyImportParams.public`

* Type: {boolean}

The `public` parameter is used to specify that the `'raw'` format key is to be interpreted as a public key. **Default:** `false`.

`NODE-SCRYPT` Algorithm

The `NODE-SCRYPT` algorithm is the common implementation of the scrypt key derivation algorithm.

Class: `NodeScryptImportParams`

`nodeScryptImportParams.name`

* Type: {string} Must be `'NODE-SCRYPT'`.

Class: `NodeScryptParams`

`nodeScryptParams.encoding`

* Type: {string} The string encoding when `salt` is a string.

`nodeScryptParams.maxmem`

* Type: {number} Memory upper bound. It is an error when (approximately) `127 * N * r > maxmem`. **Default:** `32 * 1024 * 1024`.

`nodeScryptParams.N`

* Type: {number} The CPU/memory cost parameter. Must e a power of two greater than 1. **Default:** `16384`.

`nodeScryptParams.p`

* Type: {number} Parallelization parameter. **Default:** `1`.

`nodeScryptParams.r`

* Type: {number} Block size parameter. **Default:** `8`.

`nodeScryptParams.salt`

* Type: {string|ArrayBuffer|Buffer|TypedArray|DataView}

[^1]: Non-standard Node.js-specific extension