

|                                  |               |
|----------------------------------|---------------|
| JOSE Working Group               | M. Jones      |
| Internet-Draft                   | Microsoft     |
| Intended status: Standards Track | July 30, 2012 |
| Expires: January 31, 2013        |               |

# JSON Web Algorithms (JWA)

## draft-ietf-jose-json-web-algorithms-05

### Abstract

The JSON Web Algorithms (JWA) specification enumerates cryptographic algorithms and identifiers to be used with the JSON Web Signature (JWS), JSON Web Encryption (JWE), and JSON Web Key (JWK) specifications.

### Status of this Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at <http://datatracker.ietf.org/drafts/current/>.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on January 31, 2013.

### Copyright Notice

Copyright (c) 2012 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<http://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

---

### Table of Contents

- 1. Introduction**
  - 1.1. Notational Conventions**
- 2. Terminology**
  - 2.1. Terms Incorporated from the JWS Specification**
  - 2.2. Terms Incorporated from the JWE Specification**
  - 2.3. Terms Incorporated from the JWK Specification**
  - 2.4. Defined Terms**
- 3. Cryptographic Algorithms for JWS**
  - 3.1. "alg" (Algorithm) Header Parameter Values for JWS**
  - 3.2. MAC with HMAC SHA-256, HMAC SHA-384, or HMAC SHA-512**
  - 3.3. Digital Signature with RSA SHA-256, RSA SHA-384, or RSA SHA-512**
  - 3.4. Digital Signature with ECDSA P-256 SHA-256, ECDSA P-384 SHA-384, or ECDSA P-521 SHA-512**
  - 3.5. Using the Algorithm "none"**
  - 3.6. Additional Digital Signature/MAC Algorithms and Parameters**
- 4. Cryptographic Algorithms for JWE**

- [4.1. "alg" \(Algorithm\) Header Parameter Values for JWE](#)
- [4.2. "enc" \(Encryption Method\) Header Parameter Values for JWE](#)
- [4.3. "int" \(Integrity Algorithm\) Header Parameter Values for JWE](#)
- [4.4. "kdf" \(Key Derivation Function\) Header Parameter Values for JWE](#)
- [4.5. Key Encryption with RSAES-PKCS1-V1\\_5](#)
- [4.6. Key Encryption with RSAES OAEP](#)
- [4.7. Key Encryption with AES Key Wrap](#)
- [4.8. Direct Encryption with a Shared Symmetric Key](#)
- [4.9. Key Agreement with Elliptic Curve Diffie-Hellman Ephemeral Static \(ECDH-ES\)](#)
- [4.10. Plaintext Encryption with AES CBC Mode](#)
- [4.11. Plaintext Encryption with AES GCM](#)
- [4.12. Integrity Calculation with HMAC SHA-256, HMAC SHA-384, or HMAC SHA-512](#)
- [4.13. Key Derivation with Concat KDF and SHA-256, SHA-384, or SHA-512](#)
- [4.14. Additional Encryption Algorithms and Parameters](#)
- [5. Cryptographic Algorithms for JWK](#)
  - [5.1. "alg" \(Algorithm Family\) Parameter Values for JWK](#)
  - [5.2. JWK Parameters for Elliptic Curve Keys](#)
    - [5.2.1. "crv" \(Curve\) Parameter](#)
    - [5.2.2. "x" \(X Coordinate\) Parameter](#)
    - [5.2.3. "y" \(Y Coordinate\) Parameter](#)
  - [5.3. JWK Parameters for RSA Keys](#)
    - [5.3.1. "mod" \(Modulus\) Parameter](#)
    - [5.3.2. "exp" \(Exponent\) Parameter](#)
  - [5.4. Additional Key Algorithm Families and Parameters](#)
- [6. IANA Considerations](#)
  - [6.1. JSON Web Signature and Encryption Algorithms Registry](#)
    - [6.1.1. Registration Template](#)
    - [6.1.2. Initial Registry Contents](#)
  - [6.2. JSON Web Key Algorithm Families Registry](#)
    - [6.2.1. Registration Template](#)
    - [6.2.2. Initial Registry Contents](#)
  - [6.3. JSON Web Key Parameters Registration](#)
    - [6.3.1. Registry Contents](#)
- [7. Security Considerations](#)
- [8. Open Issues](#)
- [9. References](#)
  - [9.1. Normative References](#)
  - [9.2. Informative References](#)
- [Appendix A. Digital Signature/MAC Algorithm Identifier Cross-Reference](#)
- [Appendix B. Encryption Algorithm Identifier Cross-Reference](#)
- [Appendix C. Acknowledgements](#)
- [Appendix D. Document History](#)
- [§ Author's Address](#)

---

## 1. Introduction

TOC

The JSON Web Algorithms (JWA) specification enumerates cryptographic algorithms and identifiers to be used with the JSON Web Signature (JWS) [JWS], JSON Web Encryption (JWE) [JWE], and JSON Web Key (JWK) [JWK] specifications. All these specifications utilize JavaScript Object Notation (JSON) [RFC4627] based data structures. This specification also describes the semantics and operations that are specific to these algorithms and algorithm families.

Enumerating the algorithms and identifiers for them in this specification, rather than in the JWS, JWE, and JWK specifications, is intended to allow them to remain unchanged in the face of changes in the set of required, recommended, optional, and deprecated algorithms over time.

---

### 1.1. Notational Conventions

TOC

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in Key words for use in RFCs to Indicate Requirement Levels [\[RFC2119\]](#).

---

## 2. Terminology TOC

---

### 2.1. Terms Incorporated from the JWS Specification TOC

These terms defined by the JSON Web Signature (JWS) [\[JWS\]](#) specification are incorporated into this specification:

**JSON Web Signature (JWS)**

A data structure cryptographically securing a JWS Header and a JWS Payload with a JWS Signature value.

**JWS Header**

A string representing a JavaScript Object Notation (JSON) [\[RFC4627\]](#) object that describes the digital signature or MAC operation applied to create the JWS Signature value.

**JWS Payload**

The bytes to be secured - a.k.a., the message. The payload can contain an arbitrary sequence of bytes.

**JWS Signature**

A byte array containing the cryptographic material that secures the contents of the JWS Header and the JWS Payload.

**Base64url Encoding**

The URL- and filename-safe Base64 encoding described in [RFC 4648](#) [RFC4648], Section 5, with the (non URL-safe) '=' padding characters omitted, as permitted by Section 3.2. (See Appendix C of [\[JWS\]](#) for notes on implementing base64url encoding without padding.)

**Encoded JWS Header**

Base64url encoding of the bytes of the UTF-8 [\[RFC3629\]](#) representation of the JWS Header.

**Encoded JWS Payload**

Base64url encoding of the JWS Payload.

**Encoded JWS Signature**

Base64url encoding of the JWS Signature.

**JWS Secured Input**

The concatenation of the Encoded JWS Header, a period ('.') character, and the Encoded JWS Payload.

**Collision Resistant Namespace**

A namespace that allows names to be allocated in a manner such that they are highly unlikely to collide with other names. For instance, collision resistance can be achieved through administrative delegation of portions of the namespace or through use of collision-resistant name allocation functions. Examples of Collision Resistant Namespaces include: Domain Names, Object Identifiers (OIDs) as defined in the ITU-T X.660 and X.670 Recommendation series, and Universally Unique IDentifiers (UUIDs) [\[RFC4122\]](#). When using an administratively delegated namespace, the definer of a name needs to take reasonable precautions to ensure they are in control of the portion of the namespace they use to define the name.

---

### 2.2. Terms Incorporated from the JWE Specification TOC

These terms defined by the JSON Web Encryption (JWE) [\[JWE\]](#) specification are incorporated into this specification:

**JSON Web Encryption (JWE)**

A data structure representing an encrypted version of a Plaintext. The structure consists of four parts: the JWE Header, the JWE Encrypted Key, the JWE Ciphertext,

and the JWE Integrity Value.

**Plaintext**  
The bytes to be encrypted - a.k.a., the message. The plaintext can contain an arbitrary sequence of bytes.

**Ciphertext**  
The encrypted version of the Plaintext.

**Content Encryption Key (CEK)**  
A symmetric key used to encrypt the Plaintext for the recipient to produce the Ciphertext.

**Content Integrity Key (CIK)**  
A key used with a MAC function to ensure the integrity of the Ciphertext and the parameters used to create it.

**Content Master Key (CMK)**  
A key from which the CEK and CIK are derived. When key wrapping or key encryption are employed, the CMK is randomly generated and encrypted to the recipient as the JWE Encrypted Key. When key agreement is employed, the CMK is the result of the key agreement algorithm.

**JWE Header**  
A string representing a JSON object that describes the encryption operations applied to create the JWE Encrypted Key, the JWE Ciphertext, and the JWE Integrity Value.

**JWE Encrypted Key**  
When key wrapping or key encryption are employed, the Content Master Key (CMK) is encrypted with the intended recipient's key and the resulting encrypted content is recorded as a byte array, which is referred to as the JWE Encrypted Key. Otherwise, when key agreement is employed, the JWE Encrypted Key is the empty byte array.

**JWE Ciphertext**  
A byte array containing the Ciphertext.

**JWE Integrity Value**  
A byte array containing a MAC value that ensures the integrity of the Ciphertext and the parameters used to create it.

**Encoded JWE Header**  
Base64url encoding of the bytes of the UTF-8 **[RFC3629]** representation of the JWE Header.

**Encoded JWE Encrypted Key**  
Base64url encoding of the JWE Encrypted Key.

**Encoded JWE Ciphertext**  
Base64url encoding of the JWE Ciphertext.

**Encoded JWE Integrity Value**  
Base64url encoding of the JWE Integrity Value.

**AEAD Algorithm**  
An Authenticated Encryption with Associated Data (AEAD) **[RFC5116]** encryption algorithm is one that provides an integrated content integrity check. AES Galois/Counter Mode (GCM) is one such algorithm.

---

## 2.3. Terms Incorporated from the JWK Specification

TOC

These terms defined by the JSON Web Key (JWK) **[JWK]** specification are incorporated into this specification:

**JSON Web Key (JWK)**  
A JSON data structure that represents a public key.

**JSON Web Key Set (JWK Set)**  
A JSON object that contains an array of JWKs as a member.

---

## 2.4. Defined Terms

TOC

These terms are defined for use by this specification:

**Header Parameter Name**  
The name of a member of the JSON object representing a JWS Header or JWE Header.

## Header Parameter Value

The value of a member of the JSON object representing a JWS Header or JWE Header.

---

### 3. Cryptographic Algorithms for JWS

TOC

JWS uses cryptographic algorithms to digitally sign or create a Message Authentication Codes (MAC) of the contents of the JWS Header and the JWS Payload. The use of the following algorithms for producing JWSs is defined in this section.

---

#### 3.1. "alg" (Algorithm) Header Parameter Values for JWS

TOC

The table below is the set of `alg` (algorithm) header parameter values defined by this specification for use with JWS, each of which is explained in more detail in the following sections:

| alg Parameter Value | Digital Signature or MAC Algorithm                 | Implementation Requirements |
|---------------------|--|-----------------------------|
| HS256               | HMAC using SHA-256 hash algorithm                  | REQUIRED                    |
| HS384               | HMAC using SHA-384 hash algorithm                  | OPTIONAL                    |
| HS512               | HMAC using SHA-512 hash algorithm                  | OPTIONAL                    |
| RS256               | RSASSA using SHA-256 hash algorithm                | RECOMMENDED                 |
| RS384               | RSASSA using SHA-384 hash algorithm                | OPTIONAL                    |
| RS512               | RSASSA using SHA-512 hash algorithm                | OPTIONAL                    |
| ES256               | ECDSA using P-256 curve and SHA-256 hash algorithm | RECOMMENDED+                |
| ES384               | ECDSA using P-384 curve and SHA-384 hash algorithm | OPTIONAL                    |
| ES512               | ECDSA using P-521 curve and SHA-512 hash algorithm | OPTIONAL                    |
| none                | No digital signature or MAC value included         | REQUIRED                    |

All the names are short because a core goal of JWS is for the representations to be compact. However, there is no a priori length restriction on `alg` values.

The use of "+" in the Implementation Requirements indicates that the requirement strength is likely to be increased in a future version of the specification.

See **Appendix A** for a table cross-referencing the digital signature and MAC `alg` (algorithm) values used in this specification with the equivalent identifiers used by other standards and software packages.

---

#### 3.2. MAC with HMAC SHA-256, HMAC SHA-384, or HMAC SHA-512

TOC

Hash-based Message Authentication Codes (HMACs) enable one to use a secret plus a cryptographic hash function to generate a Message Authentication Code (MAC). This can be used to demonstrate that the MAC matches the hashed content, in this case the JWS Secured Input, which therefore demonstrates that whoever generated the MAC was in possession of the secret. The means of exchanging the shared key is outside the scope of this specification.

The algorithm for implementing and validating HMACs is provided in **RFC 2104** [RFC2104]. This section defines the use of the HMAC SHA-256, HMAC SHA-384, and HMAC SHA-512 functions [**SHS**]. The `alg` (algorithm) header parameter values `HS256`, `HS384`, and `HS512` are used in the JWS Header to indicate that the Encoded JWS Signature contains a base64url encoded HMAC value using the respective hash function.

A key of the same size as the hash output (for instance, 256 bits for [HS256](#)) or larger MUST be used with this algorithm.

The HMAC SHA-256 MAC is generated per RFC 2104, using SHA-256 as the hash algorithm "H", using the bytes of the ASCII [\[USASCII\]](#) representation of the JWS Secured Input as the "text" value, and using the shared key. The HMAC output value is the JWS Signature. The JWS signature is base64url encoded to produce the Encoded JWS Signature.

The HMAC SHA-256 MAC for a JWS is validated by computing an HMAC value per RFC 2104, using SHA-256 as the hash algorithm "H", using the bytes of the ASCII representation of the received JWS Secured input as the "text" value, and using the shared key. This computed HMAC value is then compared to the result of base64url decoding the received Encoded JWS signature. Alternatively, the computed HMAC value can be base64url encoded and compared to the received Encoded JWS Signature, as this comparison produces the same result as comparing the unencoded values. In either case, if the values match, the HMAC has been validated. If the validation fails, the JWS MUST be rejected.

Securing content with the HMAC SHA-384 and HMAC SHA-512 algorithms is performed identically to the procedure for HMAC SHA-256 - just using the corresponding hash algorithm with correspondingly larger minimum key sizes and result values: 384 bits each for HMAC SHA-384 and 512 bits each for HMAC SHA-512.

---

### 3.3. Digital Signature with RSA SHA-256, RSA SHA-384, or RSA SHA-512

TOC

This section defines the use of the RSASSA-PKCS1-V1\_5 digital signature algorithm as defined in Section 8.2 of [RFC 3447](#) [RFC3447], (commonly known as PKCS #1), using SHA-256, SHA-384, or SHA-512 [\[SHS\]](#) as the hash functions. The `alg` (algorithm) header parameter values [RS256](#), [RS384](#), and [RS512](#) are used in the JWS Header to indicate that the Encoded JWS Signature contains a base64url encoded RSA digital signature using the respective hash function.

A key of size 2048 bits or larger MUST be used with these algorithms.

The RSA SHA-256 digital signature is generated as follows:

1. Generate a digital signature of the bytes of the ASCII representation of the JWS Secured Input using RSASSA-PKCS1-V1\_5-SIGN and the SHA-256 hash function with the desired private key. The output will be a byte array.
2. Base64url encode the resulting byte array.

The output is the Encoded JWS Signature for that JWS.

The RSA SHA-256 digital signature for a JWS is validated as follows:

1. Take the Encoded JWS Signature and base64url decode it into a byte array. If decoding fails, the JWS MUST be rejected.
2. Submit the bytes of the ASCII representation of the JWS Secured Input and the public key corresponding to the private key used by the signer to the RSASSA-PKCS1-V1\_5-VERIFY algorithm using SHA-256 as the hash function.
3. If the validation fails, the JWS MUST be rejected.

Signing with the RSA SHA-384 and RSA SHA-512 algorithms is performed identically to the procedure for RSA SHA-256 - just using the corresponding hash algorithm with correspondingly larger result values: 384 bits for RSA SHA-384 and 512 bits for RSA SHA-512.

---

### 3.4. Digital Signature with ECDSA P-256 SHA-256, ECDSA P-384 SHA-384, or ECDSA P-521 SHA-512

TOC

The Elliptic Curve Digital Signature Algorithm (ECDSA) [\[DSS\]](#) provides for the use of Elliptic Curve cryptography, which is able to provide equivalent security to RSA cryptography but using shorter key sizes and with greater processing speed. This means that ECDSA digital

signatures will be substantially smaller in terms of length than equivalently strong RSA digital signatures.

This specification defines the use of ECDSA with the P-256 curve and the SHA-256 cryptographic hash function, ECDSA with the P-384 curve and the SHA-384 hash function, and ECDSA with the P-521 curve and the SHA-512 hash function. The P-256, P-384, and P-521 curves are defined in **[DSS]**. The `alg` (algorithm) header parameter values `ES256`, `ES384`, and `ES512` are used in the JWS Header to indicate that the Encoded JWS Signature contains a base64url encoded ECDSA P-256 SHA-256, ECDSA P-384 SHA-384, or ECDSA P-521 SHA-512 digital signature, respectively.

The ECDSA P-256 SHA-256 digital signature is generated as follows:

1. Generate a digital signature of the bytes of the ASCII representation of the JWS Secured Input using ECDSA P-256 SHA-256 with the desired private key. The output will be the pair (R, S), where R and S are 256 bit unsigned integers.
2. Turn R and S into byte arrays in big endian order, with each array being 32 bytes long. The array representations MUST not be shortened to omit any leading zero bytes contained in the values.
3. Concatenate the two byte arrays in the order R and then S. (Note that many ECDSA implementations will directly produce this concatenation as their output.)
4. Base64url encode the resulting 64 byte array.

The output is the Encoded JWS Signature for the JWS.

The ECDSA P-256 SHA-256 digital signature for a JWS is validated as follows:

1. Take the Encoded JWS Signature and base64url decode it into a byte array. If decoding fails, the JWS MUST be rejected.
2. The output of the base64url decoding MUST be a 64 byte array. If decoding does not result in a 64 byte array, the JWS MUST be rejected.
3. Split the 64 byte array into two 32 byte arrays. The first array will be R and the second S (with both being in big endian byte order).
4. Submit the bytes of the ASCII representation of the JWS Secured Input R, S and the public key (x, y) to the ECDSA P-256 SHA-256 validator.
5. If the validation fails, the JWS MUST be rejected.

Note that ECDSA digital signature contains a value referred to as K, which is a random number generated for each digital signature instance. This means that two ECDSA digital signatures using exactly the same input parameters will output different signature values because their K values will be different. A consequence of this is that one cannot validate an ECDSA signature by recomputing the signature and comparing the results.

Signing with the ECDSA P-384 SHA-384 and ECDSA P-521 SHA-512 algorithms is performed identically to the procedure for ECDSA P-256 SHA-256 - just using the corresponding hash algorithm with correspondingly larger result values. For ECDSA P-384 SHA-384, R and S will be 384 bits each, resulting in a 96 byte array. For ECDSA P-521 SHA-512, R and S will be 521 bits each, resulting in a 132 byte array.

---

### 3.5. Using the Algorithm "none" TOC

JWSs MAY also be created that do not provide integrity protection. Such a JWS is called a "Plaintext JWS". Plaintext JWSs MUST use the `alg` value `none`, and are formatted identically to other JWSs, but with an empty JWS Signature value.

---

### 3.6. Additional Digital Signature/MAC Algorithms and Parameters TOC

Additional algorithms MAY be used to protect JWSs with corresponding `alg` (algorithm) header parameter values being defined to refer to them. New `alg` header parameter values SHOULD either be registered in the IANA JSON Web Signature and Encryption Algorithms registry **Section 6.1** or be a URI that contains a Collision Resistant Namespace. In particular, it is permissible to use the algorithm identifiers defined in **XML DSIG** [RFC3275], **XML DSIG 2.0** [W3C.CR-xmlsig-core2-20120124], and related specifications as `alg` values.

As indicated by the common registry, JWSs and JWEs share a common `alg` value space. The values used by the two specifications MUST be distinct, as the `alg` value MAY be used to determine whether the object is a JWS or JWE.

Likewise, additional reserved header parameter names MAY be defined via the IANA JSON Web Signature and Encryption Header Parameters registry [JWS]. As indicated by the common registry, JWSs and JWEs share a common header parameter space; when a parameter is used by both specifications, its usage must be compatible between the specifications.

## 4. Cryptographic Algorithms for JWE

TOC

JWE uses cryptographic algorithms to encrypt the Content Master Key (CMK) and the Plaintext. This section specifies a set of specific algorithms for these purposes.

### 4.1. "alg" (Algorithm) Header Parameter Values for JWE

TOC

The table below is the set of `alg` (algorithm) header parameter values that are defined by this specification for use with JWE. These algorithms are used to encrypt the CMK, producing the JWE Encrypted Key, or to use key agreement to agree upon the CMK.

| alg Parameter Value | Key Encryption or Agreement Algorithm   | Implementation Requirements |
|---------------------|---|-----------------------------|
| RSA1_5              | RSAES-PKCS1-V1_5 [RFC3447]  | REQUIRED                    |
| RSA-OAEP            | RSAES using Optimal Asymmetric Encryption Padding (OAEP) [RFC3447], with the default parameters specified by RFC 3447 in Section A.2.1  | OPTIONAL                    |
| A128KW              | Advanced Encryption Standard (AES) Key Wrap Algorithm [RFC3394] using 128 bit keys  | RECOMMENDED                 |
| A256KW              | AES Key Wrap Algorithm using 256 bit keys   | RECOMMENDED                 |
| dir                 | Direct use of a shared symmetric key as the Content Master Key (CMK) for the block encryption step (rather than using the symmetric key to wrap the CMK)  | RECOMMENDED                 |
| ECDH-ES             | Elliptic Curve Diffie-Hellman Ephemeral Static [RFC6090] key agreement using the Concat KDF, as defined in Section 5.8.1 of [NIST.800-56A], where the Digest Method is SHA-256 and all OtherInfo parameters are the empty bit string, with the agreed-upon key being used directly as the Content Master Key (CMK) (rather than being used to wrap the CMK) | RECOMMENDED+                |
| ECDH-ES+A128KW      | Elliptic Curve Diffie-Hellman Ephemeral Static key agreement per ECDH-ES, but where the agreed-upon key is used to wrap the Content Master Key (CMK) with the A128KW function (rather than being used directly as the CMK)  | RECOMMENDED                 |
| ECDH-ES+A256KW      | Elliptic Curve Diffie-Hellman Ephemeral Static key agreement per ECDH-ES, but where the agreed-upon key is used to wrap the Content Master Key (CMK) with the A256KW function (rather than being used directly as the CMK)  | RECOMMENDED                 |

The use of "+" in the Implementation Requirements indicates that the requirement strength is likely to be increased in a future version of the specification.

### 4.2. "enc" (Encryption Method) Header Parameter Values for JWE

TOC

The table below is the set of `enc` (encryption method) header parameter values that are



defined by this specification for use with JWE. These algorithms are used to encrypt the Plaintext, which produces the Ciphertext.

| enc Parameter Value | Block Encryption Algorithm  | Implementation Requirements |
|---------------------|---|-----------------------------|
| A128CBC             | Advanced Encryption Standard (AES) in Cipher Block Chaining (CBC) mode with PKCS #5 padding [AES] [NIST.800-38A] using 128 bit keys | REQUIRED                    |
| A256CBC             | AES in CBC mode with PKCS #5 padding using 256 bit keys   | REQUIRED                    |
| A128GCM             | AES in Galois/Counter Mode (GCM) [AES] [NIST.800-38D] using 128 bit keys  | RECOMMENDED                 |
| A256GCM             | AES GCM using 256 bit keys  | RECOMMENDED                 |

All the names are short because a core goal of JWE is for the representations to be compact. However, there is no a priori length restriction on `alg` values.

See **Appendix B** for a table cross-referencing the encryption `alg` (algorithm) and `enc` (encryption method) values used in this specification with the equivalent identifiers used by other standards and software packages.

### 4.3. "int" (Integrity Algorithm) Header Parameter Values for JWE

TOC

The table below is the set of `int` (integrity algorithm) header parameter values defined by this specification for use with JWE. Note that these are the HMAC SHA subset of the `alg` (algorithm) header parameter values defined for use with JWS **Section 3.1**.

| int Parameter Value | Algorithm                         | Implementation Requirements |
|---------------------|-----------------------------------|-----------------------------|
| HS256               | HMAC using SHA-256 hash algorithm | REQUIRED                    |
| HS384               | HMAC using SHA-384 hash algorithm | OPTIONAL                    |
| HS512               | HMAC using SHA-512 hash algorithm | OPTIONAL                    |

### 4.4. "kdf" (Key Derivation Function) Header Parameter Values for JWE

TOC

The table below is the set of `kdf` (key derivation function) header parameter values defined by this specification for use with JWE.

| kdf Parameter Value | Algorithm   | Implementation Requirements |
|---------------------|---|-----------------------------|
| CS256               | Concat KDF, as defined in Section 5.8.1 of [NIST.800-56A], with parameters per <b>Section 4.13</b> , using SHA-256 as the digest method | REQUIRED                    |
| CS384               | Concat KDF with parameters per <b>Section 4.13</b> , using SHA-384 as the digest method   | OPTIONAL                    |
| CS512               | Concat KDF with parameters per <b>Section 4.13</b> , using SHA-512 as the digest method   | OPTIONAL                    |

### 4.5. Key Encryption with RSAES-PKCS1-V1\_5

TOC

This section defines the specifics of encrypting a JWE CMK with RSAES-PKCS1-V1\_5 [RFC3447]. The `alg` header parameter value `RSA1_5` is used in this case.

A key of size 2048 bits or larger MUST be used with this algorithm.

---

## 4.6. Key Encryption with RSAES OAEP

TOC

This section defines the specifics of encrypting a JWE CMK with RSAES using Optimal Asymmetric Encryption Padding (OAEP) [RFC3447], with the default parameters specified by RFC 3447 in Section A.2.1. The `alg` header parameter value `RSA-OAEP` is used in this case.

A key of size 2048 bits or larger MUST be used with this algorithm.

---

## 4.7. Key Encryption with AES Key Wrap

TOC

This section defines the specifics of encrypting a JWE CMK with the Advanced Encryption Standard (AES) Key Wrap Algorithm [RFC3394] using 128 or 256 bit keys. The `alg` header parameter values `A128KW` or `A256KW` are used in this case.

---

## 4.8. Direct Encryption with a Shared Symmetric Key

TOC

This section defines the specifics of directly performing symmetric key encryption without performing a key wrapping step. In this case, the shared symmetric key is used directly as the Content Master Key (CMK) value for the `enc` algorithm. An empty byte array is used as the JWE Encrypted Key value. The `alg` header parameter value `dir` is used in this case.

---

## 4.9. Key Agreement with Elliptic Curve Diffie-Hellman Ephemeral Static (ECDH-ES)

TOC

This section defines the specifics of key agreement with Elliptic Curve Diffie-Hellman Ephemeral Static [RFC6090], and using the Concat KDF, as defined in Section 5.8.1 of [NIST.800-56A], where the Digest Method is SHA-256 and all OtherInfo parameters are the empty bit string. The key agreement result can be used in one of two ways: (1) directly as the Content Master Key (CMK) for the `enc` algorithm, or (2) as a symmetric key used to wrap the CMK with either the `A128KW` or `A256KW` algorithms. The `alg` header parameter values `ECDH-ES`, `ECDH-ES+A128KW`, and `ECDH-ES+A256KW` are respectively used in this case.

In the direct case, the output of the Concat KDF MUST be a key of the same length as that used by the `enc` algorithm; in this case, the empty byte array is used as the JWE Encrypted Key value. In the key wrap case, the output of the Concat KDF MUST be a key of the length needed for the specified key wrap algorithm, either 128 or 256 bits respectively.

A new `epk` (ephemeral public key) value MUST be generated for each key agreement transaction.

---

## 4.10. Plaintext Encryption with AES CBC Mode

TOC

This section defines the specifics of encrypting the JWE Plaintext with Advanced Encryption Standard (AES) in Cipher Block Chaining (CBC) mode with PKCS #5 padding [AES] [NIST.800-38A] using 128 or 256 bit keys. The `enc` header parameter values `A128CBC` or `A256CBC` are used in this case.

Use of an initialization vector of size 128 bits is REQUIRED with this algorithm.

---

## 4.11. Plaintext Encryption with AES GCM

TOC

This section defines the specifics of encrypting the JWE Plaintext with Advanced Encryption Standard (AES) in Galois/Counter Mode (GCM) **[AES] [NIST.800-38D]** using 128 or 256 bit keys. The `enc` header parameter values `A128GCM` or `A256GCM` are used in this case.

Use of an initialization vector of size 96 bits is REQUIRED with this algorithm.

The "additional authenticated data" parameter is used to secure the header and key values, as specified for AEAD algorithms in Section 5 of **[JWE]**.

The requested size of the "authentication tag" output MUST be 128 bits, regardless of the key size.

As GCM is an AEAD algorithm, the JWE Integrity Value is set to be the "authentication tag" value produced by the encryption.

---

#### 4.12. Integrity Calculation with HMAC SHA-256, HMAC SHA-384, or HMAC SHA-512

TOC

This section defines the specifics of computing a JWE Integrity Value with HMAC SHA-256, HMAC SHA-384, or HMAC SHA-512 **[SHS]**. Other than as stated below, these computations are performed identically to those specified in **Section 3.2**.

A key of the same size as the hash output (for instance, 256 bits for `HS256`) MUST be used with this algorithm.

Per Section 9 of **[JWE]**, the JWS Secured Input value used contains the header, encrypted key, and ciphertext.

---

#### 4.13. Key Derivation with Concat KDF and SHA-256, SHA-384, or SHA-512

TOC

The key derivation process derives CEK and CIK values from the CMK. It uses as a primitive a Key Derivation Function (KDF) which notionally takes three arguments:

MasterKey:

The master key used to compute the individual use keys

Label:

The use key label, used to differentiate individual use keys

Length:

The desired length of the use key

This section defines the specifics of using the Concat KDF, as defined in Section 5.8.1 of **[NIST.800-56A]**, where the Digest Method is one of SHA-256, SHA-384, or SHA-512, the `SuppPubInfo` parameter is the Label, and the remaining `OtherInfo` parameters are the empty bit string.

The `kdf` (key derivation function) header parameter values `CS256`, `CS384`, and `CS512` are respectively used in the JWE Header to indicate the use of the Concat KDF as above with the respective digest methods. If the `kdf` header parameter is omitted when an AEAD `enc` algorithm is not used, this is equivalent to specifying use of the `CS256` key derivation function.

To compute the CEK from the CMK, the ASCII label "Encryption" ([69, 110, 99, 114, 121, 112, 116, 105, 111, 110]) is used. Use the key size for the `enc` algorithm as the CEK desired key length.

To compute the CIK from the CMK, the ASCII label "Integrity" ([73, 110, 116, 101, 103, 114, 105, 116, 121]) is used. Use the minimum key size for the `int` algorithm (for instance, 256 bits for `HS256`) as the CIK desired key length.

---

#### 4.14. Additional Encryption Algorithms and Parameters

TOC

Additional algorithms MAY be used to protect JWEs with corresponding `alg` (algorithm), `enc` (encryption method), and `int` (integrity algorithm) header parameter values being defined to refer to them. New `alg`, `enc`, and `int` header parameter values SHOULD either be registered in the IANA JSON Web Signature and Encryption Algorithms registry **Section 6.1** or be a URI that contains a Collision Resistant Namespace. In particular, it is permissible to use the algorithm identifiers defined in **XML Encryption** [W3C.REC-xmlenc-core-20021210], **XML Encryption 1.1** [W3C.CR-xmlenc-core1-20120313], and related specifications as `alg`, `enc`, and `int` values.

As indicated by the common registry, JWSs and JWEs share a common `alg` value space. The values used by the two specifications MUST be distinct, as the `alg` value MAY be used to determine whether the object is a JWS or JWE.

Likewise, additional reserved header parameter names MAY be defined via the IANA JSON Web Signature and Encryption Header Parameters registry **[JWS]**. As indicated by the common registry, JWSs and JWEs share a common header parameter space; when a parameter is used by both specifications, its usage must be compatible between the specifications.

---

## 5. Cryptographic Algorithms for JWK TOC

A JSON Web Key (JWK) **[JWK]** is a JavaScript Object Notation (JSON) **[RFC4627]** data structure that represents a public key. A JSON Web Key Set (JWK Set) is a JSON data structure for representing a set of JWKs. This section specifies a set of algorithm families to be used for those public keys and the algorithm family specific parameters for representing those keys.

---

### 5.1. "alg" (Algorithm Family) Parameter Values for JWK TOC

The table below is the set of `alg` (algorithm family) parameter values that are defined by this specification for use in JWKs.

| alg Parameter Value | Algorithm Family                       | Implementation Requirements |
|---------------------|--|-----------------------------|
| EC                  | Elliptic Curve <b>[DSS]</b> key family | RECOMMENDED+                |
| RSA                 | RSA <b>[RFC3447]</b> key family        | REQUIRED                    |

All the names are short because a core goal of JWK is for the representations to be compact. However, there is no a priori length restriction on `alg` values.

The use of "+" in the Implementation Requirements indicates that the requirement strength is likely to be increased in a future version of the specification.

---

### 5.2. JWK Parameters for Elliptic Curve Keys TOC

JWKs can represent Elliptic Curve **[DSS]** keys. In this case, the `alg` member value MUST be `EC`. Furthermore, these additional members MUST be present:

---

#### 5.2.1. "crv" (Curve) Parameter TOC

The `crv` (curve) member identifies the cryptographic curve used with the key. Curve values from **[DSS]** used by this specification are:

- P-256
- P-384
- P-521

Additional `crv` values MAY be used, provided they are understood by implementations using that Elliptic Curve key. The `crv` value is a case sensitive string.

---

### 5.2.2. "x" (X Coordinate) Parameter

TOC

The `x` (x coordinate) member contains the x coordinate for the elliptic curve point. It is represented as the base64url encoding of the coordinate's big endian representation as a byte array. The array representation MUST not be shortened to omit any leading zero bytes contained in the value. For instance, when representing 521 bit integers, the byte array to be base64url encoded MUST contain 66 bytes, including any leading zero bytes.

---

### 5.2.3. "y" (Y Coordinate) Parameter

TOC

The `y` (y coordinate) member contains the y coordinate for the elliptic curve point. It is represented as the base64url encoding of the coordinate's big endian representation as a byte array. The array representation MUST not be shortened to omit any leading zero bytes contained in the value. For instance, when representing 521 bit integers, the byte array to be base64url encoded MUST contain 66 bytes, including any leading zero bytes.

---

## 5.3. JWK Parameters for RSA Keys

TOC

JWKs can represent RSA **[RFC3447]** keys. In this case, the `alg` member value MUST be `RSA`. Furthermore, these additional members MUST be present:

---

### 5.3.1. "mod" (Modulus) Parameter

TOC

The `mod` (modulus) member contains the modulus value for the RSA public key. It is represented as the base64url encoding of the value's unsigned big endian representation as a byte array. The array representation MUST not be shortened to omit any leading zero bytes. For instance, when representing 2048 bit integers, the byte array to be base64url encoded MUST contain 256 bytes, including any leading zero bytes.

---

### 5.3.2. "exp" (Exponent) Parameter

TOC

The `exp` (exponent) member contains the exponent value for the RSA public key. It is represented as the base64url encoding of the value's unsigned big endian representation as a byte array. The array representation MUST utilize the minimum number of bytes to represent the value. For instance, when representing the value 65537, the byte array to be base64url encoded MUST consist of the three bytes [1, 0, 1].

---

## 5.4. Additional Key Algorithm Families and Parameters

TOC

Public keys using additional algorithm families MAY be represented using JWK data structures with corresponding `alg` (algorithm family) parameter values being defined to refer to them. New `alg` parameter values SHOULD either be registered in the IANA JSON Web Key Algorithm Families registry **Section 6.2** or be a URI that contains a Collision Resistant Namespace.

Likewise, parameters for representing keys for additional algorithm families or additional key properties SHOULD either be registered in the IANA JSON Web Key Parameters registry **[JWK]** or be a URI that contains a Collision Resistant Namespace.

---

## 6. IANA Considerations

The following registration procedure is used for all the registries established by this specification.

Values are registered with a Specification Required **[RFC5226]** after a two week review period on the [TBD]@ietf.org mailing list, on the advice of one or more Designated Experts. However, to allow for the allocation of values prior to publication, the Designated Expert(s) may approve registration once they are satisfied that such a specification will be published.

Registration requests must be sent to the [TBD]@ietf.org mailing list for review and comment, with an appropriate subject (e.g., "Request for access token type: example"). [[ Note to RFC-EDITOR: The name of the mailing list should be determined in consultation with the IESG and IANA. Suggested name: jose-reg-review. ]]

Within the review period, the Designated Expert(s) will either approve or deny the registration request, communicating this decision to the review list and IANA. Denials should include an explanation and, if applicable, suggestions as to how to make the request successful.

IANA must only accept registry updates from the Designated Expert(s), and should direct all requests for registration to the review mailing list.

---

### 6.1. JSON Web Signature and Encryption Algorithms Registry

This specification establishes the IANA JSON Web Signature and Encryption Algorithms registry for values of the JWS and JWE **alg** (algorithm), **enc** (encryption method), and **int** (integrity algorithm) header parameters. The registry records the algorithm name, the algorithm usage locations from the set **alg**, **enc**, and **int**, implementation requirements, and a reference to the specification that defines it. The same algorithm name may be registered multiple times, provided that the sets of usage locations are disjoint. The implementation requirements of an algorithm may be changed over time by the Designated Experts(s) as the cryptographic landscape evolves, for instance, to change the status of an algorithm to DEPRECATED, or to change the status of an algorithm from OPTIONAL to RECOMMENDED or REQUIRED.

---

#### 6.1.1. Registration Template

**Algorithm Name:**

The name requested (e.g., "example"). This name is case sensitive. Names that match other registered names in a case insensitive manner SHOULD NOT be accepted.

**Algorithm Usage Location(s):**

The algorithm usage, which must be one or more of the values **alg**, **enc**, **int**, or **kdf**.

**Implementation Requirements:**

The algorithm implementation requirements, which must be one the words REQUIRED, RECOMMENDED, OPTIONAL, or DEPRECATED. Optionally, the word may be followed by a "+" or "-". The use of "+" indicates that the requirement strength is likely to be increased in a future version of the specification. The use of "-" indicates that the requirement strength is likely to be decreased in a future version of the specification.

**Change Controller:**

For standards-track RFCs, state "IETF". For others, give the name of the responsible party. Other details (e.g., postal address, e-mail address, home page URI) may also be included.

**Specification Document(s):**

Reference to the document that specifies the parameter, preferably including a URI that can be used to retrieve a copy of the document. An indication of the relevant sections may also be included, but is not required.

## 6.1.2. Initial Registry Contents

- Algorithm Name: [HS256](#)
- Algorithm Usage Location(s): [alg](#), [int](#)
- Implementation Requirements: REQUIRED
- Change Controller: IETF
- Specification Document(s): **Section 3.1** and **Section 4.3** of [[ this document ]]
  
- Algorithm Name: [HS384](#)
- Algorithm Usage Location(s): [alg](#), [int](#)
- Implementation Requirements: OPTIONAL
- Change Controller: IETF
- Specification Document(s): **Section 3.1** and **Section 4.3** of [[ this document ]]
  
- Algorithm Name: [HS512](#)
- Algorithm Usage Location(s): [alg](#), [int](#)
- Implementation Requirements: OPTIONAL
- Change Controller: IETF
- Specification Document(s): **Section 3.1** and **Section 4.3** of [[ this document ]]
  
- Algorithm Name: [RS256](#)
- Algorithm Usage Location(s): [alg](#)
- Implementation Requirements: RECOMMENDED
- Change Controller: IETF
- Specification Document(s): **Section 3.1** of [[ this document ]]
  
- Algorithm Name: [RS384](#)
- Algorithm Usage Location(s): [alg](#)
- Implementation Requirements: OPTIONAL
- Change Controller: IETF
- Specification Document(s): **Section 3.1** of [[ this document ]]
  
- Algorithm Name: [RS512](#)
- Algorithm Usage Location(s): [alg](#)
- Implementation Requirements: OPTIONAL
- Change Controller: IETF
- Specification Document(s): **Section 3.1** of [[ this document ]]
  
- Algorithm Name: [ES256](#)
- Algorithm Usage Location(s): [alg](#)
- Implementation Requirements: RECOMMENDED+
- Change Controller: IETF
- Specification Document(s): **Section 3.1** of [[ this document ]]
  
- Algorithm Name: [ES384](#)
- Algorithm Usage Location(s): [alg](#)
- Implementation Requirements: OPTIONAL
- Change Controller: IETF
- Specification Document(s): **Section 3.1** of [[ this document ]]
  
- Algorithm Name: [ES512](#)
- Algorithm Usage Location(s): [alg](#)
- Implementation Requirements: OPTIONAL
- Change Controller: IETF
- Specification Document(s): **Section 3.1** of [[ this document ]]
  
- Algorithm Name: [none](#)
- Algorithm Usage Location(s): [alg](#)
- Implementation Requirements: REQUIRED
- Change Controller: IETF
- Specification Document(s): **Section 3.1** of [[ this document ]]
  
- Algorithm Name: [RSA1\\_5](#)
- Algorithm Usage Location(s): [alg](#)

- Implementation Requirements: REQUIRED
  - Change Controller: IETF
  - Specification Document(s): **Section 4.1** of [[ this document ]]
- Algorithm Name: [RSA-OAEP](#)
  - Algorithm Usage Location(s): [alg](#)
  - Implementation Requirements: OPTIONAL
  - Change Controller: IETF
  - Specification Document(s): **Section 4.1** of [[ this document ]]
- Algorithm Name: [A128KW](#)
  - Algorithm Usage Location(s): [alg](#)
  - Implementation Requirements: RECOMMENDED
  - Change Controller: IETF
  - Specification Document(s): **Section 4.1** of [[ this document ]]
- Algorithm Name: [A256KW](#)
  - Algorithm Usage Location(s): [alg](#)
  - Implementation Requirements: RECOMMENDED
  - Change Controller: IETF
  - Specification Document(s): **Section 4.1** of [[ this document ]]
- Algorithm Name: [dir](#)
  - Algorithm Usage Location(s): [alg](#)
  - Implementation Requirements: RECOMMENDED
  - Change Controller: IETF
  - Specification Document(s): **Section 4.1** of [[ this document ]]
- Algorithm Name: [ECDH-ES](#)
  - Algorithm Usage Location(s): [alg](#)
  - Implementation Requirements: RECOMMENDED+
  - Change Controller: IETF
  - Specification Document(s): **Section 4.1** of [[ this document ]]
- Algorithm Name: [ECDH-ES+A128KW](#)
  - Algorithm Usage Location(s): [alg](#)
  - Implementation Requirements: RECOMMENDED
  - Change Controller: IETF
  - Specification Document(s): **Section 4.1** of [[ this document ]]
- Algorithm Name: [ECDH-ES+A256KW](#)
  - Algorithm Usage Location(s): [alg](#)
  - Implementation Requirements: RECOMMENDED
  - Change Controller: IETF
  - Specification Document(s): **Section 4.1** of [[ this document ]]
- Algorithm Name: [A128CBC](#)
  - Algorithm Usage Location(s): [enc](#)
  - Implementation Requirements: REQUIRED
  - Change Controller: IETF
  - Specification Document(s): **Section 4.2** of [[ this document ]]
- Algorithm Name: [A256CBC](#)
  - Algorithm Usage Location(s): [enc](#)
  - Implementation Requirements: REQUIRED
  - Change Controller: IETF
  - Specification Document(s): **Section 4.2** of [[ this document ]]
- Algorithm Name: [A128GCM](#)
  - Algorithm Usage Location(s): [enc](#)
  - Implementation Requirements: RECOMMENDED
  - Change Controller: IETF
  - Specification Document(s): **Section 4.2** of [[ this document ]]
- Algorithm Name: [A256GCM](#)
  - Algorithm Usage Location(s): [enc](#)
  - Implementation Requirements: RECOMMENDED
  - Change Controller: IETF



- Specification Document(s): **Section 4.2** of [[ this document ]]
- Algorithm Name: [CS256](#)
- Algorithm Usage Location(s): [kdf](#)
- Implementation Requirements: REQUIRED
- Change Controller: IETF
- Specification Document(s): **Section 4.4** of [[ this document ]]
- Algorithm Name: [CS384](#)
- Algorithm Usage Location(s): [kdf](#)
- Implementation Requirements: OPTIONAL
- Change Controller: IETF
- Specification Document(s): **Section 4.4** of [[ this document ]]
- Algorithm Name: [CS512](#)
- Algorithm Usage Location(s): [kdf](#)
- Implementation Requirements: OPTIONAL
- Change Controller: IETF
- Specification Document(s): **Section 4.4** of [[ this document ]]

## 6.2. JSON Web Key Algorithm Families Registry TOC

This specification establishes the IANA JSON Web Key Algorithm Families registry for values of the JWK `alg` (algorithm family) parameter. The registry records the `alg` value and a reference to the specification that defines it. This specification registers the values defined in **Section 5.1**.

### 6.2.1. Registration Template TOC

**"alg" Parameter Value:**

The name requested (e.g., "example"). This name is case sensitive. Names that match other registered names in a case insensitive manner SHOULD NOT be accepted.

**Change Controller:**

For standards-track RFCs, state "IETF". For others, give the name of the responsible party. Other details (e.g., postal address, e-mail address, home page URI) may also be included.

**Implementation Requirements:**

The algorithm implementation requirements, which must be one the words REQUIRED, RECOMMENDED, OPTIONAL, or DEPRECATED. Optionally, the word may be followed by a "+" or "-". The use of "+" indicates that the requirement strength is likely to be increased in a future version of the specification. The use of "-" indicates that the requirement strength is likely to be decreased in a future version of the specification.

**Specification Document(s):**

Reference to the document that specifies the parameter, preferably including a URI that can be used to retrieve a copy of the document. An indication of the relevant sections may also be included, but is not required.

### 6.2.2. Initial Registry Contents TOC

- "alg" Parameter Value: [EC](#)
- Implementation Requirements: RECOMMENDED+
- Change Controller: IETF
- Specification Document(s): **Section 5.1** of [[ this document ]]
- "alg" Parameter Value: [RSA](#)
- Implementation Requirements: REQUIRED
- Change Controller: IETF
- Specification Document(s): **Section 5.1** of [[ this document ]]

---

## 6.3. JSON Web Key Parameters Registration

TOC

This specification registers the parameter names defined in [Section 5.2](#) and [Section 5.3](#) in the IANA JSON Web Key Parameters registry [\[JWK\]](#).

---

### 6.3.1. Registry Contents

TOC

- Parameter Name: [crv](#)
- Change Controller: IETF
- Specification Document(s): [Section 5.2.1](#) of [[ this document ]]
  
- Parameter Name: [x](#)
- Change Controller: IETF
- Specification Document(s): [Section 5.2.2](#) of [[ this document ]]
  
- Parameter Name: [y](#)
- Change Controller: IETF
- Specification Document(s): [Section 5.2.3](#) of [[ this document ]]
  
- Parameter Name: [mod](#)
- Change Controller: IETF
- Specification Document(s): [Section 5.3.1](#) of [[ this document ]]
  
- Parameter Name: [exp](#)
- Change Controller: IETF
- Specification Document(s): [Section 5.3.2](#) of [[ this document ]]

---

## 7. Security Considerations

TOC

All of the security issues faced by any cryptographic application must be faced by a JWS/JWE/JWK agent. Among these issues are protecting the user's private key, preventing various attacks, and helping the user avoid mistakes such as inadvertently encrypting a message for the wrong recipient. The entire list of security considerations is beyond the scope of this document, but some significant concerns are listed here.

The security considerations in [\[AES\]](#), [\[DSS\]](#), [\[JWE\]](#), [\[JWK\]](#), [\[JWS\]](#), [\[NIST.800-38A\]](#), [\[NIST.800-38D\]](#), [\[NIST.800-56A\]](#), [\[RFC2104\]](#), [\[RFC3394\]](#), [\[RFC3447\]](#), [\[RFC5116\]](#), [\[RFC6090\]](#), and [\[SHS\]](#) apply to this specification.

Eventually the algorithms and/or key sizes currently described in this specification will no longer be considered sufficiently secure and will be removed. Therefore, implementers and deployments must be prepared for this eventuality.

Algorithms of matching strength should be used together whenever possible. For instance, when AES Key Wrap is used with a given key size, using the same key size for AES CBC or GCM is recommended. Likewise, when AES CBC is used with a 128 bit key, using HMAC SHA-256 as the integrity algorithm is recommended, whereas when AES CBC is used with a 256 bit key, using HMAC SHA-512 as the integrity algorithm is recommended.

While Section 8 of RFC 3447 [\[RFC3447\]](#) explicitly calls for people not to adopt RSASSA-PKCS1 for new applications and instead requests that people transition to RSASSA-PSS, this specification does include RSASSA-PKCS1, for interoperability reasons, because it commonly implemented.

Keys used with RSAES-PKCS1-v1\_5 must follow the constraints in Section 7.2 of RFC 3447 [\[RFC3447\]](#). In particular, keys with a low public key exponent value must not be used.

Plaintext JWSs (JWSs that use the `alg` value `none`) provide no integrity protection. Thus, they must only be used in contexts where the payload is secured by means other than a digital

signature or MAC value, or need not be secured.

Receiving agents that validate signatures and sending agents that encrypt messages need to be cautious of cryptographic processing usage when validating signatures and encrypting messages using keys larger than those mandated in this specification. An attacker could send certificates with keys that would result in excessive cryptographic processing, for example, keys larger than those mandated in this specification, which could swamp the processing element. Agents that use such keys without first validating the certificate to a trust anchor are advised to have some sort of cryptographic resource management system to prevent such attacks.

---

## 8. Open Issues

TOC

[[ to be removed by the RFC editor before publication as an RFC ]]

The following items remain to be considered or done in this draft:

- Should we use the `alg` value as the AlgorithmID input to the Concat KDF when doing key agreement? Or is an AlgorithmID value unnecessary in the way that we are using Concat for key agreement?
- Similarly, should we use a combination of the `enc` and `int` values as the AlgorithmID input to the Concat KDF when doing key derivation? Or is an AlgorithmID value unnecessary in the way that we are using Concat for key derivation?
- Do we want to add the output key length to the Concat KDF input parameters?
- Do we need non-empty PartyUInfo and PartyVInfo values when using the Concat KDF for key agreement? Or given that we already require the use of a random unique Ephemeral Public Key (EPK), is this superfluous, as duplicate keys will not be generated unless a duplicate EPK is used? If we do decide we need PartyUInfo and PartyVInfo values, how can we dynamically generate them from information already carried in the header, rather than requiring that they be explicitly passed as header parameters?
- Similarly, do we need non-empty PartyUInfo and PartyVInfo values when using the Concat KDF for key derivation? Or given that we already require the use of a random unique Content Master Key (CMK), is this superfluous, as duplicate keys will not be generated unless a duplicate CMK is used? If we do decide we need PartyUInfo and PartyVInfo values, how can we dynamically generate them from information already carried in the header, rather than requiring that they be explicitly passed as header parameters?
- We use the same secret multiple times with the Concat KDF with different parameters to generate multiple derived keys. Do we want to continue this practice or instead use the KDF once to generate a big enough byte array for all derived keys, and use different parts of the byte array for each key?
- Do we want to add AES ECB as a (non-authenticated) key wrap algorithm? Is there any problem with doing key wrap without an integrity check, given that a separate integrity check already covers the wrapped key?
- Do we want the ability to perform symmetric encryption directly with a shared key, without using a separate CMK? This would save space and time in the single recipient case. It would also be parallel to this option for key agreement, which doesn't use a separate CMK.
- Do we want the ability to perform symmetric encryption directly with an agreed upon key, without using a separate CMK? The arguments for this parallel those for direct encryption using a shared key.

---

## 9. References

TOC

---

### 9.1. Normative References

TOC

- [DSS] National Institute of Standards and Technology, "[Digital Signature Standard \(DSS\)](#)," FIPS PUB 186-3, June 2009.
- [JWE] [Jones, M., Rescorla, E., and J. Hildebrand](#), "[JSON Web Encryption \(JWE\)](#)," July 2012.
- [JWK] [Jones, M.](#), "[JSON Web Key \(JWK\)](#)," July 2012.
- [JWS] [Jones, M., Bradley, J., and N. Sakimura](#), "[JSON Web Signature \(JWS\)](#)," July 2012.
- [NIST.800-38A] National Institute of Standards and Technology (NIST), "[Recommendation for Block Cipher Modes of Operation](#)," NIST PUB 800-38A, December 2001.
- [NIST.800-38D] National Institute of Standards and Technology (NIST), "[Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode \(GCM\) and GMAC](#)," NIST PUB 800-38D, December 2001.
- [NIST.800-56A] National Institute of Standards and Technology (NIST), "[Recommendation for Pair-Wise Key Establishment Schemes Using Discrete Logarithm Cryptography \(Revised\)](#)," NIST PUB 800-56A, March 2007.
- [RFC2104] [Krawczyk, H., Bellare, M., and R. Canetti](#), "[HMAC: Keyed-Hashing for Message Authentication](#)," RFC 2104, February 1997 ([TXT](#)).
- [RFC2119] [Bradner, S.](#), "[Key words for use in RFCs to Indicate Requirement Levels](#)," BCP 14, RFC 2119, March 1997 ([TXT](#), [HTML](#), [XML](#)).
- [RFC3394] Schaad, J. and R. Housley, "[Advanced Encryption Standard \(AES\) Key Wrap Algorithm](#)," RFC 3394, September 2002 ([TXT](#)).
- [RFC3447] Jonsson, J. and B. Kaliski, "[Public-Key Cryptography Standards \(PKCS\) #1: RSA Cryptography Specifications Version 2.1](#)," RFC 3447, February 2003 ([TXT](#)).
- [RFC3629] Yergeau, F., "[UTF-8, a transformation format of ISO 10646](#)," STD 63, RFC 3629, November 2003 ([TXT](#)).
- [RFC4627] Crockford, D., "[The application/json Media Type for JavaScript Object Notation \(JSON\)](#)," RFC 4627, July 2006 ([TXT](#)).
- [RFC4648] Josefsson, S., "[The Base16, Base32, and Base64 Data Encodings](#)," RFC 4648, October 2006 ([TXT](#)).
- [RFC5116] McGrew, D., "[An Interface and Algorithms for Authenticated Encryption](#)," RFC 5116, January 2008 ([TXT](#)).
- [RFC5226] Narten, T. and H. Alvestrand, "[Guidelines for Writing an IANA Considerations Section in RFCs](#)," BCP 26, RFC 5226, May 2008 ([TXT](#)).
- [RFC6090] McGrew, D., Igoe, K., and M. Salter, "[Fundamental Elliptic Curve Cryptography Algorithms](#)," RFC 6090, February 2011 ([TXT](#)).
- [SHS] National Institute of Standards and Technology, "[Secure Hash Standard \(SHS\)](#)," FIPS PUB 180-3, October 2008.
- [USASCII] American National Standards Institute, "Coded Character Set -- 7-bit American Standard Code for Information Interchange," ANSI X3.4, 1986.

## 9.2. Informative References

TOC

- [CanvasApp] Facebook, "[Canvas Applications](#)," 2010.
- [I-D.rescorla-jsms] Rescorla, E. and J. Hildebrand, "[JavaScript Message Security Format](#)," draft-rescorla-jsms-00 (work in progress), March 2011 ([TXT](#)).
- [JCA] Oracle, "[Java Cryptography Architecture](#)," 2011.
- [JSE] Bradley, J. and N. Sakimura (editor), "[JSON Simple Encryption](#)," September 2010.
- [JSS] Bradley, J. and N. Sakimura (editor), "[JSON Simple Sign](#)," September 2010.
- [MagicSignatures] Panzer (editor), J., Laurie, B., and D. Balfanz, "[Magic Signatures](#)," January 2011.
- [RFC3275] Eastlake, D., Reagle, J., and D. Solo, "[\(Extensible Markup Language\) XML-Signature Syntax and Processing](#)," RFC 3275, March 2002 ([TXT](#)).
- [RFC4122] [Leach, P., Mealling, M., and R. Salz](#), "[A Universally Unique Identifier \(UUID\) URN Namespace](#)," RFC 4122, July 2005 ([TXT](#), [HTML](#), [XML](#)).
- [W3C.CR-xmldsig-core2-20120124] Solo, D., Datta, P., Hirsch, F., Cantor, S., Reagle, J., Roessler, T., Eastlake, D., and K. Yiu, "[XML Signature Syntax and Processing Version 2.0](#)," World Wide Web Consortium CR CR-xmldsig-core2-20120124, January 2012 ([HTML](#)).
- [W3C.CR-xmlenc-core1-20120313] Eastlake, D., Reagle, J., Hirsch, F., and T. Roessler, "[XML Encryption Syntax and Processing Version 1.1](#)," World Wide Web Consortium CR CR-xmlenc-core1-20120313, March 2012 ([HTML](#)).
- [W3C.REC-xmlenc-core-20021210] Eastlake, D. and J. Reagle, "[XML Encryption Syntax and Processing](#)," World Wide Web Consortium Recommendation REC-xmlenc-core-20021210, December 2002 ([HTML](#)).

## Appendix A. Digital Signature/MAC Algorithm Identifier Cross-Reference

TOC

This appendix contains a table cross-referencing the digital signature and MAC alg (algorithm) values used in this specification with the equivalent identifiers used by other standards and software packages. See [XML DSIG](#) [RFC3275], [XML DSIG 2.0](#) [W3C.CR-xmldsig-core2-20120124], and [Java Cryptography Architecture](#) [JCA] for more information about the names defined by those documents.

| Algorithm               | JWS   | XML DSIG  | JCA        | OID                |
|-------------------------|-------|---|------------|--------------------|
| HMAC using SHA-256 hash | HS256 | <a href="http://www.w3.org/2001/04/xmldsig-more#hmac-sha256">http://www.w3.org/2001/04/xmldsig-more#hmac-sha256</a> | HmacSHA256 | 1.2.840.113549.2.9 |

|  |       |   |                 |                       |
|--|-------|---|-----------------|-----------------------|
| algorithm  |       |   |                 |                       |
| HMAC using<br>SHA-384<br>hash<br>algorithm                     | HS384 | <a href="http://www.w3.org/2001/04/xmldsig-more#hmac-sha384">http://www.w3.org/2001/04/xmldsig-more#hmac-sha384</a>   | HmacSHA384      | 1.2.840.113549.2.10   |
| HMAC using<br>SHA-512<br>hash<br>algorithm                     | HS512 | <a href="http://www.w3.org/2001/04/xmldsig-more#hmac-sha512">http://www.w3.org/2001/04/xmldsig-more#hmac-sha512</a>   | HmacSHA512      | 1.2.840.113549.2.11   |
| RSASSA<br>using SHA-<br>256 hash<br>algorithm                  | RS256 | <a href="http://www.w3.org/2001/04/xmldsig-more#rsa-sha256">http://www.w3.org/2001/04/xmldsig-more#rsa-sha256</a>     | SHA256withRSA   | 1.2.840.113549.1.1.11 |
| RSASSA<br>using SHA-<br>384 hash<br>algorithm                  | RS384 | <a href="http://www.w3.org/2001/04/xmldsig-more#rsa-sha384">http://www.w3.org/2001/04/xmldsig-more#rsa-sha384</a>     | SHA384withRSA   | 1.2.840.113549.1.1.12 |
| RSASSA<br>using SHA-<br>512 hash<br>algorithm                  | RS512 | <a href="http://www.w3.org/2001/04/xmldsig-more#rsa-sha512">http://www.w3.org/2001/04/xmldsig-more#rsa-sha512</a>     | SHA512withRSA   | 1.2.840.113549.1.1.13 |
| ECDSA using<br>P-256 curve<br>and SHA-256<br>hash<br>algorithm | ES256 | <a href="http://www.w3.org/2001/04/xmldsig-more#ecdsa-sha256">http://www.w3.org/2001/04/xmldsig-more#ecdsa-sha256</a> | SHA256withECDSA | 1.2.840.10045.4.3.2   |
| ECDSA using<br>P-384 curve<br>and SHA-384<br>hash<br>algorithm | ES384 | <a href="http://www.w3.org/2001/04/xmldsig-more#ecdsa-sha384">http://www.w3.org/2001/04/xmldsig-more#ecdsa-sha384</a> | SHA384withECDSA | 1.2.840.10045.4.3.3   |
| ECDSA using<br>P-521 curve<br>and SHA-512<br>hash<br>algorithm | ES512 | <a href="http://www.w3.org/2001/04/xmldsig-more#ecdsa-sha512">http://www.w3.org/2001/04/xmldsig-more#ecdsa-sha512</a> | SHA512withECDSA | 1.2.840.10045.4.3.4   |

## Appendix B. Encryption Algorithm Identifier Cross-Reference

TOC

This appendix contains a table cross-referencing the `alg` (algorithm) and `enc` (encryption method) values used in this specification with the equivalent identifiers used by other standards and software packages. See **XML Encryption** [W3C.REC-xmlenc-core-20021210], **XML Encryption 1.1** [W3C.CR-xmlenc-core1-20120313], and **Java Cryptography Architecture** [JCA] for more information about the names defined by those documents.

| Algorithm  | JWE          | XML ENC   | JCA                                     |
|--|--------------|---|---|
| RSAES-PKCS1-V1_5   | RSA1_5       | <a href="http://www.w3.org/2001/04/xmlenc#rsa-1_5">http://www.w3.org/2001/04/xmlenc#rsa-1_5</a>                               | RSA/ECB/PKCS1Padding                    |
| RSAES using<br>Optimal<br>Asymmetric<br>Encryption Padding<br>(OAEP)                 | RSA-<br>OAEP | <a href="http://www.w3.org/2001/04/xmlenc#rsa-&lt;br/&gt;oaep-mgf1p">http://www.w3.org/2001/04/xmlenc#rsa-<br/>oaep-mgf1p</a> | RSA/ECB/OAEPWithSHA-<br>1AndMGF1Padding |
| Elliptic Curve Diffie-<br>Hellman Ephemeral<br>Static                                | ECDH-ES      | <a href="http://www.w3.org/2009/xmlenc11#ECDH-&lt;br/&gt;ES">http://www.w3.org/2009/xmlenc11#ECDH-<br/>ES</a>                 |   |
| Advanced<br>Encryption Standard<br>(AES) Key Wrap<br>Algorithm using 128<br>bit keys | A128KW       | <a href="http://www.w3.org/2001/04/xmlenc#kw-&lt;br/&gt;aes128">http://www.w3.org/2001/04/xmlenc#kw-<br/>aes128</a>           |   |

|   |         |   |                      |
|---|---------|---|----------------------|
| AES key wrap<br>Algorithm using 256<br>bit keys   | A256KW  | <a href="http://www.w3.org/2001/04/xmlenc#kw-aes256">http://www.w3.org/2001/04/xmlenc#kw-aes256</a>   |                      |
| AES in Cipher Block<br>Chaining (CBC)<br>mode with PKCS #5<br>padding using 128<br>bit keys | A128CBC | <a href="http://www.w3.org/2001/04/xmlenc#aes128-cbc">http://www.w3.org/2001/04/xmlenc#aes128-cbc</a> | AES/CBC/PKCS5Padding |
| AES in CBC mode<br>with PKCS #5<br>padding using 256<br>bit keys                            | A256CBC | <a href="http://www.w3.org/2001/04/xmlenc#aes256-cbc">http://www.w3.org/2001/04/xmlenc#aes256-cbc</a> | AES/CBC/PKCS5Padding |
| AES in<br>Galois/Counter<br>Mode (GCM) using<br>128 bit keys                                | A128GCM | <a href="http://www.w3.org/2009/xmlenc11#aes128-gcm">http://www.w3.org/2009/xmlenc11#aes128-gcm</a>   | AES/GCM/NoPadding    |
| AES GCM using 256<br>bit keys   | A256GCM | <a href="http://www.w3.org/2009/xmlenc11#aes256-gcm">http://www.w3.org/2009/xmlenc11#aes256-gcm</a>   | AES/GCM/NoPadding    |

## Appendix C. Acknowledgements

TOC

Solutions for signing and encrypting JSON content were previously explored by **Magic Signatures** [MagicSignatures], **JSON Simple Sign** [JSS], **Canvas Applications** [CanvasApp], **JSON Simple Encryption** [JSE], and **JavaScript Message Security Format** [I-D.rescorla-jsms], all of which influenced this draft. Dirk Balfanz, John Bradley, Yaron Y. Goland, John Panzer, Nat Sakimura, and Paul Tarjan all made significant contributions to the design of this specification and its related specifications.

## Appendix D. Document History

TOC

[[ to be removed by the RFC editor before publication as an RFC ]]

-05

- Support both direct encryption using a shared or agreed upon symmetric key, and the use of a shared or agreed upon symmetric key to key wrap the CMK. Specifically, added the `alg` values `dir`, `ECDH-ES+A128KW`, and `ECDH-ES+A256KW` to finish filling in this set of capabilities.
- Updated open issues.

-04

- Added text requiring that any leading zero bytes be retained in base64url encoded key value representations for fixed-length values.
- Added this language to Registration Templates: "This name is case sensitive. Names that match other registered names in a case insensitive manner SHOULD NOT be accepted."
- Described additional open issues.
- Applied editorial suggestions.

-03

- Always use a 128 bit "authentication tag" size for AES GCM, regardless of the key size.
- Specified that use of a 128 bit IV is REQUIRED with AES CBC. It was previously RECOMMENDED.
- Removed key size language for ECDSA algorithms, since the key size is implied by the algorithm being used.
- Stated that the `int` key size must be the same as the hash output size (and not larger, as was previously allowed) so that its size is defined for key generation purposes.
- Added the `kdf` (key derivation function) header parameter to provide crypto

agility for key derivation. The default KDF remains the Concat KDF with the SHA-256 digest function.

- Clarified that the `mod` and `exp` values are unsigned.
- Added Implementation Requirements columns to algorithm tables and Implementation Requirements entries to algorithm registries.
- Changed AES Key Wrap to RECOMMENDED.
- Moved registries JSON Web Signature and Encryption Header Parameters and JSON Web Signature and Encryption Type Values to the JWS specification.
- Moved JSON Web Key Parameters registry to the JWK specification.
- Changed registration requirements from RFC Required to Specification Required with Expert Review.
- Added Registration Template sections for defined registries.
- Added Registry Contents sections to populate registry values.
- No longer say "the UTF-8 representation of the JWS Secured Input (which is the same as the ASCII representation)". Just call it "the ASCII representation of the JWS Secured Input".
- Added "Collision Resistant Namespace" to the terminology section.
- Numerous editorial improvements.

-02

- For AES GCM, use the "additional authenticated data" parameter to provide integrity for the header, encrypted key, and ciphertext and use the resulting "authentication tag" value as the JWE Integrity Value.
- Defined minimum required key sizes for algorithms without specified key sizes.
- Defined KDF output key sizes.
- Specified the use of PKCS #5 padding with AES-CBC.
- Generalized text to allow key agreement to be employed as an alternative to key wrapping or key encryption.
- Clarified that ECDH-ES is a key agreement algorithm.
- Required implementation of AES-128-KW and AES-256-KW.
- Removed the use of `A128GCM` and `A256GCM` for key wrapping.
- Removed `A512KW` since it turns out that it's not a standard algorithm.
- Clarified the relationship between `typ` header parameter values and MIME types.
- Generalized language to refer to Message Authentication Codes (MACs) rather than Hash-based Message Authentication Codes (HMACs) unless in a context specific to HMAC algorithms.
- Established registries: JSON Web Signature and Encryption Header Parameters, JSON Web Signature and Encryption Algorithms, JSON Web Signature and Encryption "typ" Values, JSON Web Key Parameters, and JSON Web Key Algorithm Families.
- Moved algorithm-specific definitions from JWK to JWA.
- Reformatted to give each member definition its own section heading.

-01

- Moved definition of "alg":"none" for JWSs here from the JWT specification since this functionality is likely to be useful in more contexts than just for JWTs.
- Added Advanced Encryption Standard (AES) Key Wrap Algorithm using 512 bit keys (`A512KW`).
- Added text "Alternatively, the Encoded JWS Signature MAY be base64url decoded to produce the JWS Signature and this value can be compared with the computed HMAC value, as this comparison produces the same result as comparing the encoded values".
- Corrected the Magic Signatures reference.
- Made other editorial improvements suggested by JOSE working group participants.

-00

- Created the initial IETF draft based upon draft-jones-json-web-signature-04 and draft-jones-json-web-encryption-02 with no normative changes.
- Changed terminology to no longer call both digital signatures and HMACs "signatures".

## Author's Address

Michael B. Jones  
Microsoft

**Email:** [mbj@microsoft.com](mailto:mbj@microsoft.com)

**URI:** <http://self-issued.info/>