JSON Web Algorithms (JWA)
draft-ietf-jose-json-web-algorithms-12

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

Copyright Notice

Table of Contents

1. Introduction

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 key types.

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

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

2.1. Terms Incorporated from the JWS Specification

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

JSON Web Signature (JWS) A data structure representing a digitally
   signed or MACed message. The structure represents three values:
   the JWS Header, the JWS Payload, and the JWS Signature.

JSON Text Object A UTF-8 [RFC3629] encoded text string representing
   a JSON object; the syntax of JSON objects is defined in Section
   2.2 of [RFC4627].

JWS Header A JSON Text Object (or JSON Text Objects, when using the
   JWS JSON Serialization) that describes the digital signature or
   MAC operation applied to create the JWS Signature value. The
   members of the JWS Header object(s) are Header Parameters.

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

JWS Signature   A sequence of octets containing the cryptographic
   material that ensures the integrity of the JWS Protected Header
   and the JWS Payload. The JWS Signature value is a digital
   signature or MAC value calculated over the JWS Signing Input using
   the parameters specified in the JWS Header.

JWS Protected Header   A JSON Text Object that contains the portion of
   the JWS Header that is integrity protected. For the JWS Compact
   Serialization, this comprises the entire JWS Header. For the JWS
   JSON Serialization, this is one component of the JWS Header.

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 JWS Protected Header.

Encoded JWS Payload   Base64url encoding of the JWS Payload.

Encoded JWS Signature   Base64url encoding of the JWS Signature.

JWS Signing 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

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
   message. The structure represents five values: the JWE Header,
   the JWE Encrypted Key, the JWE Initialization Vector, the JWE
   Ciphertext, and the JWE Authentication Tag.

Authenticated Encryption An Authenticated Encryption algorithm is
   one that provides an integrated content integrity check.
   Authenticated Encryption algorithms accept two inputs, the
   Plaintext and the Additional Authenticated Data value, and produce
   two outputs, the Ciphertext and the Authentication Tag value. AES
   Galois/Counter Mode (GCM) is one such algorithm.

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

Ciphertext An encrypted representation of the Plaintext.

Additional Authenticated Data (AAD) An input to an Authenticated
   Encryption operation that is integrity protected but not
   encrypted.

Authentication Tag An output of an Authenticated Encryption
   operation that ensures the integrity of the Ciphertext and the
   Additional Authenticated Data. Note that some algorithms may not
   use an Authentication Tag, in which case this value is the empty
   octet sequence.

Content Encryption Key (CEK) A symmetric key for the Authenticated
   Encryption algorithm used to encrypt the Plaintext for the
   recipient to produce the Ciphertext and the Authentication Tag.

JWE Header A JSON Text Object (or JSON Text Objects, when using the
   JWE JSON Serialization) that describes the encryption operations
   applied to create the JWE Encrypted Key, the JWE Ciphertext, and
   the JWE Authentication Tag. The members of the JWE Header
   object(s) are Header Parameters.

JWE Encrypted Key The result of encrypting the Content Encryption
   Key (CEK) with the intended recipient's key using the specified
   algorithm. Note that for some algorithms, the JWE Encrypted Key
   value is specified as being the empty octet sequence.

JWE Initialization Vector A sequence of octets containing the
   Initialization Vector used when encrypting the Plaintext. Note
   that some algorithms may not use an Initialization Vector, in
   which case this value is the empty octet sequence.

   JWE Ciphertext A sequence of octets containing the Ciphertext for a
      JWE.

   JWE Authentication Tag A sequence of octets containing the
      Authentication Tag for a JWE.

   JWE Protected Header A JSON Text Object that contains the portion of
      the JWE Header that is integrity protected. For the JWE Compact
      Serialization, this comprises the entire JWE Header. For the JWE
      JSON Serialization, this is one component of the JWE Header.

   Encoded JWE Header Base64url encoding of the JWE Protected Header.

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

   Encoded JWE Initialization Vector Base64url encoding of the JWE
      Initialization Vector.

   Encoded JWE Ciphertext Base64url encoding of the JWE Ciphertext.

   Encoded JWE Authentication Tag Base64url encoding of the JWE
      Authentication Tag.

   Key Management Mode A method of determining the Content Encryption
      Key (CEK) value to use. Each algorithm used for determining the
      CEK value uses a specific Key Management Mode. Key Management
      Modes employed by this specification are Key Encryption, Key
      Wrapping, Direct Key Agreement, Key Agreement with Key Wrapping,
      and Direct Encryption.

   Key Encryption A Key Management Mode in which the Content Encryption
      Key (CEK) value is encrypted to the intended recipient using an
      asymmetric encryption algorithm.

   Key Wrapping A Key Management Mode in which the Content Encryption
      Key (CEK) value is encrypted to the intended recipient using a
      symmetric key wrapping algorithm.

   Direct Key Agreement A Key Management Mode in which a key agreement
      algorithm is used to agree upon the Content Encryption Key (CEK)
      value.

   Key Agreement with Key Wrapping A Key Management Mode in which a key
      agreement algorithm is used to agree upon a symmetric key used to
      encrypt the Content Encryption Key (CEK) value to the intended
      recipient using a symmetric key wrapping algorithm.

   Direct Encryption A Key Management Mode in which the Content
      Encryption Key (CEK) value used is the secret symmetric key value
      shared between the parties.

2.3. Terms Incorporated from the JWK Specification

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

   JSON Web Key (JWK) A JSON object that represents a cryptographic
      key.

   JSON Web Key Set (JWK Set) A JSON object that contains an array of
      JWKs as the value of its "keys" member.

2.4. Defined Terms

   These terms are defined for use by this specification:

   Header Parameter A name/value pair that is member of a JWS Header or
      JWE Header.

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

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

3. Cryptographic Algorithms for JWS

   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

   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       | Digital Signature or MAC Algorithm   | Implementation |
| Parameter |                                      | Requirements   |
| Value     |                                      |                |
+-----------+--------------------------------------+----------------+
| 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-PKCS-v1_5 using SHA-256 hash  | RECOMMENDED    |
|           | algorithm                            |                |
| RS384     | RSASSA-PKCS-v1_5 using SHA-384 hash  | OPTIONAL       |
|           | algorithm                            |                |
| RS512     | RSASSA-PKCS-v1_5 using SHA-512 hash  | OPTIONAL       |
|           | algorithm                            |                |
| ES256     | ECDSA using P-256 curve and SHA-256  | RECOMMENDED+   |
|           | hash algorithm                       |                |
| ES384     | ECDSA using P-384 curve and SHA-384  | OPTIONAL       |
|           | hash algorithm                       |                |
| ES512     | ECDSA using P-521 curve and SHA-512  | OPTIONAL       |
|           | hash algorithm                       |                |
| PS256     | RSASSA-PSS using SHA-256 hash        | OPTIONAL       |
|           | algorithm and MGF1 mask generation   |                |
|           | function with SHA-256                |                |
| PS512     | RSASSA-PSS using SHA-512 hash        | OPTIONAL       |
|           | algorithm and MGF1 mask generation   |                |
|           | function with SHA-512                |                |
| none      | No digital signature or MAC value    | REQUIRED       |
|           | included                             |                |
+-----------+--------------------------------------+----------------+
```

   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

   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 Signing 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 octets of the ASCII [USASCII]
representation of the JWS Signing 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 octets of the ASCII representation of the received JWS Signing
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.

An example using this algorithm is shown in Appendix A.1 of [JWS].

3.3. Digital Signature with RSASSA-PKCS1-V1_5 and SHA-256, SHA-384, or SHA-512

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 RSASSA-
PKCS1-V1_5 digital signature using the respective hash function.

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

The RSASSA-PKCS1-V1_5 SHA-256 digital signature is generated as
follows:

1. Generate a digital signature of the octets of the ASCII
    representation of the JWS Signing Input using RSASSA-PKCS1-V1_5-
    SIGN and the SHA-256 hash function with the desired private key.
    The output will be an octet sequence.

2. Base64url encode the resulting octet sequence.

The output is the Encoded JWS Signature for that JWS.

The RSASSA-PKCS1-V1_5 SHA-256 digital signature for a JWS is
validated as follows:

1. Take the Encoded JWS Signature and base64url decode it into an
    octet sequence. If decoding fails, the JWS MUST be rejected.

2. Submit the octets of the ASCII representation of the JWS Signing
    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 RSASSA-PKCS1-V1_5 SHA-384 and RSASSA-PKCS1-V1_5 SHA-
512 algorithms is performed identically to the procedure for RSASSA-
PKCS1-V1_5 SHA-256 - just using the corresponding hash algorithm with
correspondingly larger result values: 384 bits for RSASSA-PKCS1-V1_5
SHA-384 and 512 bits for RSASSA-PKCS1-V1_5 SHA-512.

An example using this algorithm is shown in Appendix A.2 of [JWS].

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

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 octets of the ASCII
    representation of the JWS Signing 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 octet sequences in big endian order, with each
    array being be 32 octets long. The array representations MUST
    NOT be shortened to omit any leading zero octets contained in the
    values.

3. Concatenate the two octet sequences 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 octet sequence.

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 an
    octet sequence. If decoding fails, the JWS MUST be rejected.

2. The output of the base64url decoding MUST be a 64 octet sequence.
    If decoding does not result in a 64 octet sequence, the JWS MUST
    be rejected.

3. Split the 64 octet sequence into two 32 octet sequences. The
    first array will be R and the second S (with both being in big
    endian octet order).

4. Submit the octets of the ASCII representation of the JWS Signing
    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 octet sequence. For ECDSA P-521 SHA-512, R and S will be 521 bits each, resulting in a 132 octet sequence.

Examples using these algorithms are shown in Appendices A.3 and A.4 of [JWS].

3.5. Digital Signature with RSASSA-PSS and SHA-256 or SHA-512

This section defines the use of the RSASSA-PSS digital signature algorithm as defined in Section 8.1 of RFC 3447 [RFC3447] with the MGF1 mask generation function, always using the same hash function for both the RSASSA-PSS hash function and the MGF1 hash function. Use of both SHA-256 and SHA-512 as these hash functions is defined. All other algorithm parameters use the defaults specified in Section A.2.3 of RFC 3447. The "alg" (algorithm) header parameter values "PS256" and "PS512" is used in the JWS Header to indicate that the Encoded JWS Signature contains a base64url encoded RSASSA-PSS digital signature using the respective hash function in both roles.

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

The RSASSA-PSS SHA-256 digital signature is generated as follows:

1. Generate a digital signature of the octets of the ASCII representation of the JWS Signing Input using RSASSA-PSS-SIGN, the SHA-256 hash function, and the MGF1 mask generation function with SHA-256 with the desired private key. The output will be an octet sequence.

2. Base64url encode the resulting octet sequence.

The output is the Encoded JWS Signature for that JWS.

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

1. Take the Encoded JWS Signature and base64url decode it into an octet sequence. If decoding fails, the JWS MUST be rejected.

2. Submit the octets of the ASCII representation of the JWS Signing Input and the public key corresponding to the private key used by the signer to the RSASSA-PSS-VERIFY algorithm using SHA-256 as the hash function and using MGF1 as the mask generation function with SHA-256.

3. If the validation fails, the JWS MUST be rejected.

Signing with the RSASSA-PSS SHA-512 algorithm is performed identically to the procedure for RSASSA-PSS SHA-256 - just using the alternative hash algorithm in both roles.

3.6. Using the Algorithm "none"

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 the empty string for its JWS Signature value.

3.7. Additional Digital Signature/MAC Algorithms and Parameters

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 value 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-xmldsig-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 can be used to determine whether the object is a JWS or JWE.

Likewise, additional reserved Header Parameter Names can 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

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

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

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 CEK, producing the JWE
Encrypted Key, or to use key agreement to agree upon the CEK.

alg Parameter Value	Key Management Algorithm	Additional Header Parameters	Implementation Requirements
RSA1_5	RSAES-PKCS1-V1_5 [RFC3447]	(none)	REQUIRED
RSA-OAEP	RSAES using Optimal Asymmetric Encryption Padding (OAEP) [RFC3447], with the default parameters specified by RFC 3447 in Section A.2.1	(none)	OPTIONAL
A128KW	Advanced Encryption Standard (AES) Key Wrap Algorithm [RFC3394] using the default initial value specified in Section 2.2.3.1 and using 128 bit keys	(none)	RECOMMENDED

	A256KW	AES Key Wrap Algorithm using the default initial value specified in Section 2.2.3.1 and using 256 bit keys	(none)	RECOMMENDED
	dir	Direct use of a shared symmetric key as the Content Encryption Key (CEK) for the content encryption step (rather than using the symmetric key to wrap the CEK)	(none)	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], with the agreed-upon key being used directly as the Content Encryption Key (CEK) (rather than being used to wrap the CEK), as specified in Section 4.7	"epk", "apu", "apv"	RECOMMENDED+

ECDH-ES+A128KW	Elliptic Curve Diffie-Hellman Ephemeral Static key agreement per "ECDH-ES" and Section 4.7, but where the agreed-upon key is used to wrap the Content Encryption Key (CEK) with the "A128KW" function (rather than being used directly as the CEK)	"epk", "apu", "apv"	RECOMMENDED
ECDH-ES+A256KW	Elliptic Curve Diffie-Hellman Ephemeral Static key agreement per "ECDH-ES" and Section 4.7, but where the agreed-upon key is used to wrap the Content Encryption Key (CEK) with the "A256KW" function (rather than being used directly as the CEK)	"epk", "apu", "apv"	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.

The Additional Header Parameters column indicates what additional Header Parameters are used by the algorithm, beyond "alg", which all use. All but "dir" and "ECDH-ES" also produce a JWE Encrypted Key value.

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

   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	Content Encryption Algorithm	Additional Header Parameters	Implementation Requirements
A128CBC-HS256	The AES_128_CBC_HMAC_SHA_256 authenticated encryption algorithm, as defined in Section 4.8.3. This algorithm uses a 256 bit key.	(none)	REQUIRED
A256CBC-HS512	The AES_256_CBC_HMAC_SHA_512 authenticated encryption algorithm, as defined in Section 4.8.4. This algorithm uses a 512 bit key.	(none)	REQUIRED
A128GCM	AES in Galois/Counter Mode (GCM) [AES] [NIST.800-38D] using 128 bit keys	(none)	RECOMMENDED
A256GCM	AES GCM using 256 bit keys	(none)	RECOMMENDED

   The Additional Header Parameters column indicates what additional
   Header Parameters are used by the algorithm, beyond "enc", which all
   use. All also use a JWE Initialization Vector value and and produce
   JWE Ciphertext and JWE Authentication Tag 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. Key Encryption with RSAES-PKCS1-V1_5

   This section defines the specifics of encrypting a JWE CEK 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.

   An example using this algorithm is shown in Appendix A.2 of [JWE].

4.4. Key Encryption with RSAES OAEP

   This section defines the specifics of encrypting a JWE CEK 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.

   An example using this algorithm is shown in Appendix A.1 of [JWE].

4.5. Key Wrapping with AES Key Wrap

   This section defines the specifics of encrypting a JWE CEK with the
   Advanced Encryption Standard (AES) Key Wrap Algorithm [RFC3394] using
   the default initial value specified in Section 2.2.3.1 using 128 or
   256 bit keys. The "alg" header parameter values "A128KW" or "A256KW"
   are respectively used in this case.

   An example using this algorithm is shown in Appendix A.3 of [JWE].

4.6. Direct Encryption with a Shared Symmetric Key

   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 Encryption
   Key (CEK) value for the "enc" algorithm. An empty octet sequence is
   used as the JWE Encrypted Key value. The "alg" header parameter
   value "dir" is used in this case.

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

   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]. The key
   agreement result can be used in one of two ways:

1. directly as the Content Encryption Key (CEK) for the "enc"
   algorithm, in the Direct Key Agreement mode, or

2. as a symmetric key used to wrap the CEK with either the "A128KW"
   or "A256KW" algorithms, in the Key Agreement with Key Wrapping
   mode.

The "alg" header parameter value "ECDH-ES" is used in the Direct Key
Agreement mode and the values "ECDH-ES+A128KW" and "ECDH-ES+A256KW"
are used in the Key Agreement with Key Wrapping mode.

In the Direct Key Agreement 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 octet sequence is used as the JWE Encrypted Key
value. In the Key Agreement with Key Wrapping case, the output of
the Concat KDF MUST be a key of the length needed for the specified
key wrapping algorithm, either 128 or 256 bits respectively.

A new ephemeral public key value MUST be generated for each key
agreement transaction.

4.7.1. Header Parameters Used for ECDH Key Agreement

The following Header Parameter Names are reserved and are used for
key agreement as defined below. They MAY also be used for other
algorithms if so specified by those algorithm parameter definitions.

4.7.1.1. "epk" (Ephemeral Public Key) Header Parameter

The "epk" (ephemeral public key) value created by the originator for
the use in key agreement algorithms. This key is represented as a
JSON Web Key [JWK] bare public key value. This Header Parameter is
REQUIRED and MUST be understood and processed by implementations when
these algorithms are used.

4.7.1.2. "apu" (Agreement PartyUInfo) Header Parameter

The "apu" (agreement PartyUInfo) value for key agreement algorithms
using it (such as "ECDH-ES"), represented as a base64url encoded
string. When used, the PartyUInfo value contains information about
the sender. Use of this Header Parameter is OPTIONAL. This Header
Parameter MUST be understood and processed by implementations when
these algorithms are used.

4.7.1.3. "apv" (Agreement PartyVInfo) Header Parameter

The "apv" (agreement PartyVInfo) value for key agreement algorithms
using it (such as "ECDH-ES"), represented as a base64url encoded

string. When used, the PartyVInfo value contains information about
the receiver. Use of this Header Parameter is OPTIONAL. This Header
Parameter MUST be understood and processed by implementations when
these algorithms are used.

4.7.2. Key Derivation for ECDH Key Agreement

The key derivation process derives the agreed upon key from the
shared secret Z established through the ECDH algorithm, per Section
6.2.2.2 of [NIST.800-56A].

Key derivation is performed using the Concat KDF, as defined in
Section 5.8.1 of [NIST.800-56A], where the Digest Method is SHA-256.
The Concat KDF parameters are set as follows:

Z This is set to the representation of the shared secret Z as an
   octet sequence.

keydatalen This is set to the number of bits in the desired output
   key. For "ECDH-ES", this is length of the key used by the "enc"
   algorithm. For "ECDH-ES+A128KW", and "ECDH-ES+A256KW", this is
   128 and 256, respectively.

AlgorithmID In the Direct Key Agreement case, this is set to the
   octets of the UTF-8 representation of the "enc" header parameter
   value. In the Key Agreement with Key Wrapping case, this is set
   to the octets of the UTF-8 representation of the "alg" header
   parameter value.

PartyUInfo The PartyUInfo value is of the form Datalen || Data,
   where Data is a variable-length string of zero or more octets, and
   Datalen is a fixed-length, big endian 32 bit counter that
   indicates the length (in octets) of Data, with || being
   concatenation. If an "apu" (agreement PartyUInfo) header
   parameter is present, Data is set to the result of base64url
   decoding the "apu" value and Datalen is set to the number of
   octets in Data. Otherwise, Datalen is set to 0 and Data is set to
   the empty octet sequence.

PartyVInfo The PartyVInfo value is of the form Datalen || Data,
   where Data is a variable-length string of zero or more octets, and
   Datalen is a fixed-length, big endian 32 bit counter that
   indicates the length (in octets) of Data, with || being
   concatenation. If an "apv" (agreement PartyVInfo) header
   parameter is present, Data is set to the result of base64url
   decoding the "apv" value and Datalen is set to the number of
   octets in Data. Otherwise, Datalen is set to 0 and Data is set to
   the empty octet sequence.

   SuppPubInfo This is set to the keydatalen represented as a 32 bit
      big endian integer.

   SuppPrivInfo This is set to the empty octet sequence.

   Note: The Diffie-Hellman Key Agreement Method [RFC2631] uses a key
   derivation function similar to the Concat KDF, but with fewer
   parameters. Rather than having separate PartyUInfo and PartyVInfo
   parameters, it uses a single PartyAInfo parameter, which is a random
   string provided by the sender, that contains 512 bits of information,
   when provided. It has no suppPrivInfo parameter. Should it be
   appropriate for the application, key agreement can be performed in a
   manner akin to RFC 2631 by using the PartyAInfo value as the "apu"
   (Agreement PartyUInfo) header parameter value, when provided, and by
   using no "apv" (Agreement PartyVInfo) header parameter.

4.8. AES_CBC_HMAC_SHA2 Algorithms

   This section defines a family of authenticated encryption algorithms
   built using a composition of Advanced Encryption Standard (AES) in
   Cipher Block Chaining (CBC) mode with PKCS #5 padding [AES]
   [NIST.800-38A] operations and HMAC [RFC2104] [SHS] operations. This
   algorithm family is called AES_CBC_HMAC_SHA2. It also defines two
   instances of this family, one using 128 bit CBC keys and HMAC SHA-256
   and the other using 256 bit CBC keys and HMAC SHA-512. Test cases
   for these algorithms can be found in Appendix C.

   These algorithms are based upon Authenticated Encryption with AES-CBC
   and HMAC-SHA [I-D.mcgrew-aead-aes-cbc-hmac-sha2], performing the same
   cryptographic computations, but with the Initialization Vector and
   Authentication Tag values remaining separate, rather than being
   concatenated with the Ciphertext value in the output representation.
   This algorithm family is a generalization of the algorithm family in
   [I-D.mcgrew-aead-aes-cbc-hmac-sha2], and can be used to implement
   those algorithms.

4.8.1. Conventions Used in Defining AES_CBC_HMAC_SHA2

   We use the following notational conventions.

      CBC-PKCS5-ENC(X, P) denotes the AES CBC encryption of P using PKCS
      #5 padding using the cipher with the key X.

      MAC(Y, M) denotes the application of the Message Authentication
      Code (MAC) to the message M, using the key Y.

      The concatenation of two octet strings A and B is denoted as
      A || B.

4.8.2. Generic AES_CBC_HMAC_SHA2 Algorithm

   This section defines AES_CBC_HMAC_SHA2 in a manner that is
   independent of the AES CBC key size or hash function to be used.
   Section 4.8.2.1 and Section 4.8.2.2 define the generic encryption and
   decryption algorithms. Section 4.8.3 and Section 4.8.4 define
   instances of AES_CBC_HMAC_SHA2 that specify those details.

4.8.2.1. AES_CBC_HMAC_SHA2 Encryption

   The authenticated encryption algorithm takes as input four octet
   strings: a secret key K, a plaintext P, associated data A, and an
   initialization vector IV. The authenticated ciphertext value E and
   the authentication tag value T are provided as outputs. The data in
   the plaintext are encrypted and authenticated, and the associated
   data are authenticated, but not encrypted.

   The encryption process is as follows, or uses an equivalent set of
   steps:

   1. The secondary keys MAC_KEY and ENC_KEY are generated from the
       input key K as follows. Each of these two keys is an octet
       string.

          MAC_KEY consists of the initial MAC_KEY_LEN octets of K, in
          order.

          ENC_KEY consists of the final ENC_KEY_LEN octets of K, in
          order.

       Here we denote the number of octets in the MAC_KEY as
       MAC_KEY_LEN, and the number of octets in ENC_KEY as ENC_KEY_LEN;
       the values of these parameters are specified by the AEAD
       algorithms (in Section 4.8.3 and Section 4.8.4). The number of
       octets in the input key K is the sum of MAC_KEY_LEN and
       ENC_KEY_LEN. When generating the secondary keys from K, MAC_KEY
       and ENC_KEY MUST NOT overlap. Note that the MAC key comes before
       the encryption key in the input key K; this is in the opposite
       order of the algorithm names in the identifier
       "AES_CBC_HMAC_SHA2".

   2. The Initialization Vector (IV) used is a 128 bit value generated
       randomly or pseudorandomly for use in the cipher.

   3. The plaintext is CBC encrypted using PKCS #5 padding using
       ENC_KEY as the key, and the IV. We denote the ciphertext output
       from this step as E.

4. The octet string AL is equal to the number of bits in A expressed
    as a 64-bit unsigned integer in network byte order.

5. A message authentication tag T is computed by applying HMAC
    [RFC2104] to the following data, in order:

        the associated data A,

        the initialization vector IV,

        the ciphertext E computed in the previous step, and

        the octet string AL defined above.

    The string MAC_KEY is used as the MAC key. We denote the output
    of the MAC computed in this step as M. The first T_LEN bits of M
    are used as T.

6. The Ciphertext E and the Authentication Tag T are returned as the
    outputs of the authenticated encryption.

The encryption process can be illustrated as follows. Here K, P, A,
IV, and E denote the key, plaintext, associated data, initialization
vector, and ciphertext, respectively.

    MAC_KEY = initial MAC_KEY_LEN bytes of K,

    ENC_KEY = final ENC_KEY_LEN bytes of K,

    E = CBC-PKCS5-ENC(ENC_KEY, P),

    M = MAC(MAC_KEY, A || IV || E || AL),

    T = initial T_LEN bytes of M.

4.8.2.2. AES_CBC_HMAC_SHA2 Decryption

The authenticated decryption operation has four inputs: K, A, E, and
T as defined above. It has only a single output, either a plaintext
value P or a special symbol FAIL that indicates that the inputs are
not authentic. The authenticated decryption algorithm is as follows,
or uses an equivalent set of steps:

1. The secondary keys MAC_KEY and ENC_KEY are generated from the
    input key K as in Step 1 of Section 4.8.2.1.

2. The integrity and authenticity of A and E are checked by
    computing an HMAC with the inputs as in Step 5 of

Section 4.8.2.1. The value T, from the previous step, is
compared to the first MAC_KEY length bits of the HMAC output. If
those values are identical, then A and E are considered valid,
and processing is continued. Otherwise, all of the data used in
the MAC validation are discarded, and the AEAD decryption
operation returns an indication that it failed, and the operation
halts. (But see Section 10 of [JWE] for security considerations
on thwarting timing attacks.)

3. The value E is decrypted and the PKCS #5 padding is removed. The
    value IV is used as the initialization vector. The value ENC_KEY
    is used as the decryption key.

4. The plaintext value is returned.

4.8.3. AES_128_CBC_HMAC_SHA_256

This algorithm is a concrete instantiation of the generic
AES_CBC_HMAC_SHA2 algorithm above. It uses the HMAC message
authentication code [RFC2104] with the SHA-256 hash function [SHS] to
provide message authentication, with the HMAC output truncated to 128
bits, corresponding to the HMAC-SHA-256-128 algorithm defined in
[RFC4868]. For encryption, it uses AES in the Cipher Block Chaining
(CBC) mode of operation as defined in Section 6.2 of [NIST.800-38A],
with PKCS #5 padding.

The input key K is 32 octets long.

The AES CBC IV is 16 octets long. ENC_KEY_LEN is 16 octets.

The SHA-256 hash algorithm is used in HMAC. MAC_KEY_LEN is 16
octets. The HMAC-SHA-256 output is truncated to T_LEN=16 octets, by
stripping off the final 16 octets.

4.8.4. AES_256_CBC_HMAC_SHA_512

AES_256_CBC_HMAC_SHA_512 is based on AES_128_CBC_HMAC_SHA_256, but
with the following differences:

A 256 bit AES CBC key is used instead of 128.

SHA-512 is used in HMAC instead of SHA-256.

ENC_KEY_LEN is 32 octets.

MAC_KEY_LEN is 32 octets.

The length of the input key K is 64 octets.

The HMAC SHA-512 value is truncated to T_LEN=32 octets instead of 16 octets.

4.8.5. Plaintext Encryption with AES_CBC_HMAC_SHA2

The algorithm value "A128CBC-HS256" is used as the "alg" value when using AES_128_CBC_HMAC_SHA_256 with JWE. The algorithm value "A256CBC-HS512" is used as the "alg" value when using AES_256_CBC_HMAC_SHA_512 with JWE. The Additional Authenticated Data value used is the octets of the ASCII representation of the Encoded JWE Header value. The JWE Initialization Vector value used is the IV value.

4.9. Plaintext Encryption with AES GCM

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 respectively used in this case.

The CEK is used as the encryption key.

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

The Additional Authenticated Data value used is the octets of the ASCII representation of the Encoded JWE Header value.

The requested size of the Authentication Tag output MUST be 128 bits, regardless of the key size.

The JWE Authentication Tag is set to be the Authentication Tag value produced by the encryption. During decryption, the received JWE Authentication Tag is used as the Authentication Tag value.

An example using this algorithm is shown in Appendix A.1 of [JWE].

4.10. Additional Encryption Algorithms and Parameters

Additional algorithms MAY be used to protect JWEs with corresponding "alg" (algorithm) and "enc" (encryption method) header parameter values being defined to refer to them. New "alg" and "enc" header parameter values SHOULD either be registered in the IANA JSON Web Signature and Encryption Algorithms registry Section 6.1 or be a value 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"
and "enc" 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 can be used to determine whether the
object is a JWS or JWE.

Likewise, additional reserved Header Parameter Names can 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

A JSON Web Key (JWK) [JWK] is a JavaScript Object Notation (JSON)
[RFC4627] data structure that represents a cryptographic 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 key types to be used for
those keys and the key type specific parameters for representing
those keys. Parameters are defined for public, private, and
symmetric keys.

5.1. "kty" (Key Type) Parameter Values for JWK

The table below is the set of "kty" (key type) parameter values that
are defined by this specification for use in JWKs.

kty Parameter Value	Key Type	Implementation Requirements
EC	Elliptic Curve [DSS] key type	RECOMMENDED+
RSA	RSA [RFC3447] key type	REQUIRED
oct	Octet sequence key type (used to represent symmetric keys)	RECOMMENDED+

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 "kty" 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

JWKs can represent Elliptic Curve [DSS] keys. In this case, the
"kty" member value MUST be "EC".

5.2.1. JWK Parameters for Elliptic Curve Public Keys

These members MUST be present for Elliptic Curve public keys:

5.2.1.1. "crv" (Curve) Parameter

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

o "P-256"

o "P-384"

o "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.

5.2.1.2. "x" (X Coordinate) Parameter

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 an octet sequence. The
array representation MUST NOT be shortened to omit any leading zero
octets contained in the value. For instance, when representing 521
bit integers, the octet sequence to be base64url encoded MUST contain
66 octets, including any leading zero octets.

5.2.1.3. "y" (Y Coordinate) Parameter

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 an octet sequence. The
array representation MUST NOT be shortened to omit any leading zero
octets contained in the value. For instance, when representing 521
bit integers, the octet sequence to be base64url encoded MUST contain
66 octets, including any leading zero octets.

5.2.2. JWK Parameters for Elliptic Curve Private Keys

In addition to the members used to represent Elliptic Curve public
keys, the following member MUST be present to represent Elliptic
Curve private keys:

5.2.2.1. "d" (ECC Private Key) Parameter

The "d" (ECC private key) member contains the Elliptic Curve private
key value. It is represented as the base64url encoding of the
value's unsigned big endian representation as an octet sequence. The
array representation MUST NOT be shortened to omit any leading zero
octets. For instance, when representing 521 bit integers, the octet
sequence to be base64url encoded MUST contain 66 octets, including
any leading zero octets.

5.3. JWK Parameters for RSA Keys

JWKs can represent RSA [RFC3447] keys. In this case, the "kty"
member value MUST be "RSA".

5.3.1. JWK Parameters for RSA Public Keys

These members MUST be present for RSA public keys:

5.3.1.1. "n" (Modulus) Parameter

The "n" (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 an octet sequence. The
array representation MUST NOT be shortened to omit any leading zero
octets. For instance, when representing 2048 bit integers, the octet
sequence to be base64url encoded MUST contain 256 octets, including
any leading zero octets.

5.3.1.2. "e" (Exponent) Parameter

The "e" (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 an octet sequence. The
array representation MUST utilize the minimum number of octets to
represent the value. For instance, when representing the value
65537, the octet sequence to be base64url encoded MUST consist of the
three octets [1, 0, 1].

5.3.2. JWK Parameters for RSA Private Keys

In addition to the members used to represent RSA public keys, the
following members are used to represent RSA private keys. All are
REQUIRED for RSA private keys except for "oth", which is sometimes
REQUIRED and sometimes MUST NOT be present, as described below.

5.3.2.1. "d" (Private Exponent) Parameter

The "d" (private exponent) member contains the private exponent value
for the RSA private key. It is represented as the base64url encoding
of the value's unsigned big endian representation as an octet
sequence. The array representation MUST NOT be shortened to omit any
leading zero octets. For instance, when representing 2048 bit
integers, the octet sequence to be base64url encoded MUST contain 256
octets, including any leading zero octets.

5.3.2.2. "p" (First Prime Factor) Parameter

The "p" (first prime factor) member contains the first prime factor,
a positive integer. It is represented as the base64url encoding of
the value's unsigned big endian representation as an octet sequence.

5.3.2.3. "q" (Second Prime Factor) Parameter

The "q" (second prime factor) member contains the second prime
factor, a positive integer. It is represented as the base64url
encoding of the value's unsigned big endian representation as an
octet sequence.

5.3.2.4. "dp" (First Factor CRT Exponent) Parameter

The "dp" (first factor CRT exponent) member contains the Chinese
Remainder Theorem (CRT) exponent of the first factor, a positive
integer. It is represented as the base64url encoding of the value's
unsigned big endian representation as an octet sequence.

5.3.2.5. "dq" (Second Factor CRT Exponent) Parameter

The "dq" (second factor CRT exponent) member contains the Chinese
Remainder Theorem (CRT) exponent of the second factor, a positive
integer. It is represented as the base64url encoding of the value's
unsigned big endian representation as an octet sequence.

5.3.2.6. "qi" (First CRT Coefficient) Parameter

The "dp" (first CRT coefficient) member contains the Chinese
Remainder Theorem (CRT) coefficient of the second factor, a positive

integer. It is represented as the base64url encoding of the value's
unsigned big endian representation as an octet sequence.

5.3.2.7. "oth" (Other Primes Info) Parameter

The "oth" (other primes info) member contains an array of information
about any third and subsequent primes, should they exist. When only
two primes have been used (the normal case), this parameter MUST be
omitted. When three or more primes have been used, the number of
array elements MUST be the number of primes used minus two. Each
array element MUST be an object with the following members:

5.3.2.7.1. "r" (Prime Factor)

The "r" (prime factor) parameter within an "oth" array member
represents the value of a subsequent prime factor, a positive
integer. It is represented as the base64url encoding of the value's
unsigned big endian representation as an octet sequence.

5.3.2.7.2. "d" (Factor CRT Exponent)

The "d" (Factor CRT Exponent) parameter within an "oth" array member
represents the CRT exponent of the corresponding prime factor, a
positive integer. It is represented as the base64url encoding of the
value's unsigned big endian representation as an octet sequence.

5.3.2.7.3. "t" (Factor CRT Coefficient)

The "t" (factor CRT coefficient) parameter within an "oth" array
member represents the CRT coefficient of the corresponding prime
factor, a positive integer. It is represented as the base64url
encoding of the value's unsigned big endian representation as an
octet sequence.

5.3.3. JWK Parameters for Symmetric Keys

When the JWK "kty" member value is "oct" (octet sequence), the
following member is used to represent a symmetric key (or another key
whose value is a single octet sequence):

5.3.3.1. "k" (Key Value) Parameter

The "k" (key value) member contains the value of the symmetric (or
other single-valued) key. It is represented as the base64url
encoding of the octet sequence containing the key value.

5.4. Additional Key Types and Parameters

Keys using additional key types can be represented using JWK data
structures with corresponding "kty" (key type) parameter values being
defined to refer to them. New "kty" parameter values SHOULD either
be registered in the IANA JSON Web Key Types registry Section 6.2 or
be a value that contains a Collision Resistant Namespace.

Likewise, parameters for representing keys for additional key types
or additional key properties SHOULD either be registered in the IANA
JSON Web Key Parameters registry [JWK] or be a value 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) and "enc" (encryption method) header parameters. The
registry records the algorithm name, the algorithm usage locations

   from the set "alg" and "enc", 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. 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"
      or "enc".

   Implementation Requirements:
      The algorithm implementation requirements, which must be one the
      words REQUIRED, RECOMMENDED, OPTIONAL, or DEPRECATED. Optionally,
      the word can 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,
      email address, home page URI) may also be included.

   Specification Document(s):
      Reference to the document(s) that specify the parameter,
      preferably including URI(s) that can be used to retrieve copies of
      the document(s). An indication of the relevant sections may also
      be included but is not required.

6.1.2. Initial Registry Contents

   o Algorithm Name: "HS256"
   o Algorithm Usage Location(s): "alg"
   o Implementation Requirements: REQUIRED
   o Change Controller: IETF

   o Specification Document(s): Section 3.1 of [[this document]]

   o Algorithm Name: "HS384"
   o Algorithm Usage Location(s): "alg"
   o Implementation Requirements: OPTIONAL
   o Change Controller: IETF
   o Specification Document(s): Section 3.1 of [[this document]]

   o Algorithm Name: "HS512"
   o Algorithm Usage Location(s): "alg"
   o Implementation Requirements: OPTIONAL
   o Change Controller: IETF
   o Specification Document(s): Section 3.1 of [[this document]]

   o Algorithm Name: "RS256"
   o Algorithm Usage Location(s): "alg"
   o Implementation Requirements: RECOMMENDED
   o Change Controller: IETF
   o Specification Document(s): Section 3.1 of [[this document]]

   o Algorithm Name: "RS384"
   o Algorithm Usage Location(s): "alg"
   o Implementation Requirements: OPTIONAL
   o Change Controller: IETF
   o Specification Document(s): Section 3.1 of [[this document]]

   o Algorithm Name: "RS512"
   o Algorithm Usage Location(s): "alg"
   o Implementation Requirements: OPTIONAL
   o Change Controller: IETF
   o Specification Document(s): Section 3.1 of [[this document]]

   o Algorithm Name: "ES256"
   o Algorithm Usage Location(s): "alg"
   o Implementation Requirements: RECOMMENDED+
   o Change Controller: IETF
   o Specification Document(s): Section 3.1 of [[this document]]

   o Algorithm Name: "ES384"
   o Algorithm Usage Location(s): "alg"
   o Implementation Requirements: OPTIONAL
   o Change Controller: IETF
   o Specification Document(s): Section 3.1 of [[this document]]

   o Algorithm Name: "ES512"
   o Algorithm Usage Location(s): "alg"

o Implementation Requirements: OPTIONAL
o Change Controller: IETF
o Specification Document(s): Section 3.1 of [[this document]]

o Algorithm Name: "PS256"
o Algorithm Usage Location(s): "alg"
o Implementation Requirements: RECOMMENDED
o Change Controller: IETF
o Specification Document(s): Section 3.1 of [[this document]]

o Algorithm Name: "PS512"
o Algorithm Usage Location(s): "alg"
o Implementation Requirements: RECOMMENDED
o Change Controller: IETF
o Specification Document(s): Section 3.1 of [[this document]]

o Algorithm Name: "none"
o Algorithm Usage Location(s): "alg"
o Implementation Requirements: REQUIRED
o Change Controller: IETF
o Specification Document(s): Section 3.1 of [[this document]]

o Algorithm Name: "RSA1_5"
o Algorithm Usage Location(s): "alg"
o Implementation Requirements: REQUIRED
o Change Controller: IETF
o Specification Document(s): Section 4.1 of [[this document]]

o Algorithm Name: "RSA-OAEP"
o Algorithm Usage Location(s): "alg"
o Implementation Requirements: OPTIONAL
o Change Controller: IETF
o Specification Document(s): Section 4.1 of [[this document]]

o Algorithm Name: "A128KW"
o Algorithm Usage Location(s): "alg"
o Implementation Requirements: RECOMMENDED
o Change Controller: IETF
o Specification Document(s): Section 4.1 of [[this document]]

o Algorithm Name: "A256KW"
o Algorithm Usage Location(s): "alg"
o Implementation Requirements: RECOMMENDED
o Change Controller: IETF
o Specification Document(s): Section 4.1 of [[this document]]

o Algorithm Name: "dir"
o Algorithm Usage Location(s): "alg"
o Implementation Requirements: RECOMMENDED
o Change Controller: IETF
o Specification Document(s): Section 4.1 of [[this document]]

o Algorithm Name: "ECDH-ES"
o Algorithm Usage Location(s): "alg"
o Implementation Requirements: RECOMMENDED+
o Change Controller: IETF
o Specification Document(s): Section 4.1 of [[this document]]

o Algorithm Name: "ECDH-ES+A128KW"
o Algorithm Usage Location(s): "alg"
o Implementation Requirements: RECOMMENDED
o Change Controller: IETF
o Specification Document(s): Section 4.1 of [[this document]]

o Algorithm Name: "ECDH-ES+A256KW"
o Algorithm Usage Location(s): "alg"
o Implementation Requirements: RECOMMENDED
o Change Controller: IETF
o Specification Document(s): Section 4.1 of [[this document]]

o Algorithm Name: "A128CBC-HS256"
o Algorithm Usage Location(s): "enc"
o Implementation Requirements: REQUIRED
o Change Controller: IETF
o Specification Document(s): Section 4.2 of [[this document]]

o Algorithm Name: "A256CBC-HS512"
o Algorithm Usage Location(s): "enc"
o Implementation Requirements: REQUIRED
o Change Controller: IETF
o Specification Document(s): Section 4.2 of [[this document]]

o Algorithm Name: "A128GCM"
o Algorithm Usage Location(s): "enc"
o Implementation Requirements: RECOMMENDED
o Change Controller: IETF
o Specification Document(s): Section 4.2 of [[this document]]

o Algorithm Name: "A256GCM"
o Algorithm Usage Location(s): "enc"
o Implementation Requirements: RECOMMENDED
o Change Controller: IETF

   o Specification Document(s): Section 4.2 of [[this document]]

6.2. JSON Web Key Types Registry

   This specification establishes the IANA JSON Web Key Types registry
   for values of the JWK "kty" (key type) parameter. The registry
   records the "kty" 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

   "kty" 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,
      email 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 can 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(s) that specify the parameter,
      preferably including URI(s) that can be used to retrieve copies of
      the document(s). An indication of the relevant sections may also
      be included but is not required.

6.2.2. Initial Registry Contents

   o "kty" Parameter Value: "EC"
   o Implementation Requirements: RECOMMENDED+
   o Change Controller: IETF
   o Specification Document(s): Section 5.1 of [[this document]]

   o "kty" Parameter Value: "RSA"
   o Implementation Requirements: REQUIRED

   o Change Controller: IETF
   o Specification Document(s): Section 5.1 of [[this document]]

   o "kty" Parameter Value: "oct"
   o Implementation Requirements: RECOMMENDED+
   o Change Controller: IETF
   o Specification Document(s): Section 5.1 of [[this document]]

6.3. JSON Web Key Parameters Registration

This specification registers the parameter names defined in Sections
5.2, 5.3, and 5.3.3 in the IANA JSON Web Key Parameters registry
[JWK].

6.3.1. Registry Contents

   o Parameter Name: "crv"
   o Parameter Information Class: Public
   o Change Controller: IETF
   o Specification Document(s): Section 5.2.1.1 of [[this document]]

   o Parameter Name: "x"
   o Parameter Information Class: Public
   o Change Controller: IETF
   o Specification Document(s): Section 5.2.1.2 of [[this document]]

   o Parameter Name: "y"
   o Parameter Information Class: Public
   o Change Controller: IETF
   o Specification Document(s): Section 5.2.1.3 of [[this document]]

   o Parameter Name: "d"
   o Parameter Information Class: Private
   o Change Controller: IETF
   o Specification Document(s): Section 5.2.2.1 of [[this document]]

   o Parameter Name: "n"
   o Parameter Information Class: Public
   o Change Controller: IETF
   o Specification Document(s): Section 5.3.1.1 of [[this document]]

   o Parameter Name: "e"
   o Parameter Information Class: Public
   o Change Controller: IETF
   o Specification Document(s): Section 5.3.1.2 of [[this document]]

o Parameter Name: "d"
o Parameter Information Class: Private
o Change Controller: IETF
o Specification Document(s): Section 5.3.2.1 of [[this document]]

o Parameter Name: "p"
o Parameter Information Class: Private
o Change Controller: IETF
o Specification Document(s): Section 5.3.2.2 of [[this document]]

o Parameter Name: "q"
o Parameter Information Class: Private
o Change Controller: IETF
o Specification Document(s): Section 5.3.2.3 of [[this document]]

o Parameter Name: "dp"
o Parameter Information Class: Private
o Change Controller: IETF
o Specification Document(s): Section 5.3.2.4 of [[this document]]

o Parameter Name: "dq"
o Parameter Information Class: Private
o Change Controller: IETF
o Specification Document(s): Section 5.3.2.5 of [[this document]]

o Parameter Name: "qi"
o Parameter Information Class: Private
o Change Controller: IETF
o Specification Document(s): Section 5.3.2.6 of [[this document]]

o Parameter Name: "oth"
o Parameter Information Class: Private
o Change Controller: IETF
o Specification Document(s): Section 5.3.2.7 of [[this document]]

o Parameter Name: "k"
o Parameter Information Class: Private
o Change Controller: IETF
o Specification Document(s): Section 5.3.3.1 of [[this document]]

6.4. Registration of JWE Header Parameter Names

This specification registers the Header Parameter Names defined in
Section 4.7.1 in the IANA JSON Web Signature and Encryption Header
Parameters registry [JWS].

6.4.1. Registry Contents

    o Header Parameter Name: "epk"
    o Header Parameter Usage Location(s): JWE
    o Change Controller: IETF
    o Specification Document(s): Section 4.7.1.1 of [[this document]]

    o Header Parameter Name: "apu"
    o Header Parameter Usage Location(s): JWE
    o Change Controller: IETF
    o Specification Document(s): Section 4.7.1.2 of [[this document]]

    o Header Parameter Name: "apv"
    o Header Parameter Usage Location(s): JWE
    o Change Controller: IETF
    o Specification Document(s): Section 4.7.1.3 of [[this document]]

7. Security Considerations

    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 and symmetric keys, 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 considerations 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 strengths should be used together whenever
    possible. For instance, when AES Key Wrap is used with a given key
    size, using the same key size is recommended when AES GCM is also
    used.

    While Section 8 of RFC 3447 [RFC3447] explicitly calls for people not
    to adopt RSASSA-PKCS-v1_5 for new applications and instead requests
    that people transition to RSASSA-PSS, this specification does include
    RSASSA-PKCS-v1_5, 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. References

8.1. Normative References

[AES]   National Institute of Standards and Technology (NIST),
             "Advanced Encryption Standard (AES)", FIPS PUB 197,
             November 2001.

[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)", draft-ietf-jose-json-web-encryption
             (work in progress), July 2013.

[JWK]   Jones, M., "JSON Web Key (JWK)",
             draft-ietf-jose-json-web-key (work in progress),
             July 2013.

[JWS]   Jones, M., Bradley, J., and N. Sakimura, "JSON Web
             Signature (JWS)", draft-ietf-jose-json-web-signature (work
             in progress), July 2013.

[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", NIST Special
            Publication 800-56A, Revision 2, May 2013.

[RFC2104]   Krawczyk, H., Bellare, M., and R. Canetti, "HMAC: Keyed-
            Hashing for Message Authentication", RFC 2104,
            February 1997.

[RFC2119]   Bradner, S., "Key words for use in RFCs to Indicate
            Requirement Levels", BCP 14, RFC 2119, March 1997.

[RFC3394]   Schaad, J. and R. Housley, "Advanced Encryption Standard
            (AES) Key Wrap Algorithm", RFC 3394, September 2002.

[RFC3629]   Yergeau, F., "UTF-8, a transformation format of ISO
            10646", STD 63, RFC 3629, November 2003.

[RFC4627]   Crockford, D., "The application/json Media Type for
            JavaScript Object Notation (JSON)", RFC 4627, July 2006.

[RFC4648]   Josefsson, S., "The Base16, Base32, and Base64 Data
            Encodings", RFC 4648, October 2006.

[RFC4868]   Kelly, S. and S. Frankel, "Using HMAC-SHA-256, HMAC-SHA-
            384, and HMAC-SHA-512 with IPsec", RFC 4868, May 2007.

[RFC5116]   McGrew, D., "An Interface and Algorithms for Authenticated
            Encryption", RFC 5116, January 2008.

[RFC5226]   Narten, T. and H. Alvestrand, "Guidelines for Writing an
            IANA Considerations Section in RFCs", BCP 26, RFC 5226,
            May 2008.

[RFC6090]   McGrew, D., Igoe, K., and M. Salter, "Fundamental Elliptic
            Curve Cryptography Algorithms", RFC 6090, February 2011.

[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.

8.2. Informative References

   [CanvasApp]
                  Facebook, "Canvas Applications", 2010.

   [I-D.mcgrew-aead-aes-cbc-hmac-sha2]
                  McGrew, D. and K. Paterson, "Authenticated Encryption with
                  AES-CBC and HMAC-SHA",
                  draft-mcgrew-aead-aes-cbc-hmac-sha2-01 (work in progress),
                  October 2012.

   [I-D.rescorla-jsms]
                  Rescorla, E. and J. Hildebrand, "JavaScript Message
                  Security Format", draft-rescorla-jsms-00 (work in
                  progress), March 2011.

   [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.

   [RFC2631] Rescorla, E., "Diffie-Hellman Key Agreement Method",
                  RFC 2631, June 1999.

   [RFC3275] Eastlake, D., Reagle, J., and D. Solo, "(Extensible Markup
                  Language) XML-Signature Syntax and Processing", RFC 3275,
                  March 2002.

   [RFC3447] Jonsson, J. and B. Kaliski, "Public-Key Cryptography
                  Standards (PKCS) #1: RSA Cryptography Specifications
                  Version 2.1", RFC 3447, February 2003.

   [RFC4122] Leach, P., Mealling, M., and R. Salz, "A Universally
                  Unique IDentifier (UUID) URN Namespace", RFC 4122,
                  July 2005.

   [W3C.CR-xmldsig-core2-20120124]
                  Eastlake, D., Reagle, J., Yiu, K., Solo, D., Datta, P.,

          Hirsch, F., Cantor, S., and T. Roessler, "XML Signature
          Syntax and Processing Version 2.0", World Wide Web
          Consortium CR CR-xmldsig-core2-20120124, January 2012,
          <http://www.w3.org/TR/2012/CR-xmldsig-core2-20120124>.

   [W3C.CR-xmlenc-core1-20120313]
          Eastlake, D., Reagle, J., Roessler, T., and F. Hirsch,
          "XML Encryption Syntax and Processing Version 1.1", World
          Wide Web Consortium CR CR-xmlenc-core1-20120313,
          March 2012,
          <http://www.w3.org/TR/2012/CR-xmlenc-core1-20120313>.

   [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,
          <http://www.w3.org/TR/2002/REC-xmlenc-core-20021210>.

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

   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 algorithm	HS256	http://www.w3.org/2001/04/xmldsig-more#hmac-sha256	HmacSHA256	1.2.840.113549.2.9
HMAC using SHA-384 hash algorithm	HS384	http://www.w3.org/2001/04/xmldsig-more#hmac-sha384	HmacSHA384	1.2.840.113549.2.10

HMAC using SHA-512 hash algorithm	HS512	http://www.w3.org/2001/04/xmldsig-more#hmac-sha512	HmacSHA512	1.2.840.113549.2.11
RSASSA-PKCS-v1_5using SHA-256hash algorithm	RS256	http://www.w3.org/2001/04/xmldsig-more#rsa-sha256	SHA256withRSA	1.2.840.113549.1.1.11
RSASSA-PKCS-v1_5using SHA-384hash algorithm	RS384	http://www.w3.org/2001/04/xmldsig-more#rsa-sha384	SHA384withRSA	1.2.840.113549.1.1.12
RSASSA-PKCS-v1_5using SHA-512hash algorithm	RS512	http://www.w3.org/2001/04/xmldsig-more#rsa-sha512	SHA512withRSA	1.2.840.113549.1.1.13
ECDSA using P-256 curve and SHA-256 hash algorithm	ES256	http://www.w3.org/2001/04/xmldsig-more#ecdsa-sha256	SHA256withECDSA	1.2.840.10045.4.3.2
ECDSA using P-384 curve and SHA-384 hash algorithm	ES384	http://www.w3.org/2001/04/xmldsig-more#ecdsa-sha384	SHA384withECDSA	1.2.840.10045.4.3.3

ECDSA using P-521 curve and SHA-512 hash algorithm	ES 512	http://www.w3.org/2001/04/xmldsig-more#ecdsa-sha512	SHA512withECDSA	1.2.840.10045.4.3.4
RSASSA-PSS using SHA-256hash algorithm and MGF1 mask generation functionwith SHA-256	PS 256			
RSASSA-PSS using SHA-512hash algorithm and MGF1 mask generation functionwith SHA-512	PS 512			

Appendix B. Encryption Algorithm Identifier Cross-Reference

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.

For the composite algorithms "A128CBC-HS256" and "A256CBC-HS512", the
corresponding AES CBC algorithm identifiers are listed.

Algorith m	JWE	XML ENC	JCA
RSAES-PK CS1-V1_5	RSA1_5	http://www.w3.org/2001/0 4/xmlenc#rsa-1_5	RSA/ECB/PKCS1Paddi ng
RSAES using Optimal Asymmetr ic Encrypt ion Paddin g (OAEP)	RSA-OA EP	http://www.w3.org/2001/0 4/xmlenc#rsa-oaep-mgf1p	RSA/ECB/OAEPWithSH A-1AndMGF1Padding
Elliptic Curve Diffie-H ellman Ephemer alStatic	ECDH-E S	http://www.w3.org/2009/x mlenc11#ECDH-ES	
Advanced Encrypti on Standar d(AES) Key Wra pAlgorit hmusing 128 bi t keys	A128KW	http://www.w3.org/2001/0 4/xmlenc#kw-aes128	
AES Key Wrap Algorith musing 256 bit keys	A256KW	http://www.w3.org/2001/0 4/xmlenc#kw-aes256	

AES in Cipher Block Chaining (CBC) mode with PKCS #5 padding using 128 bit keys	A128CBC-HS256	http://www.w3.org/2001/04/xmlenc#aes128-cbc	AES/CBC/PKCS5Padding
AES in CBC mode with PKCS #5 padding using 256 bit keys	A256CBC-HS512	http://www.w3.org/2001/04/xmlenc#aes256-cbc	AES/CBC/PKCS5Padding
AES in Galois/Counter Mode (GCM) using 128 bit keys	A128GCM	http://www.w3.org/2009/xmlenc11#aes128-gcm	AES/GCM/NoPadding
AES GCM using 256 bit keys	A256GCM	http://www.w3.org/2009/xmlenc11#aes256-gcm	AES/GCM/NoPadding

Appendix C. Test Cases for AES_CBC_HMAC_SHA2 Algorithms

The following test cases can be used to validate implementations of
the AES_CBC_HMAC_SHA2 algorithms defined in Section 4.8. They are
also intended to correspond to test cases that may appear in a future
version of [I-D.mcgrew-aead-aes-cbc-hmac-sha2], demonstrating that
the cryptographic computations performed are the same.

The variable names are those defined in Section 4.8. All values are
hexadecimal.

C.1. Test Cases for AES_128_CBC_HMAC_SHA_256

   AES_128_CBC_HMAC_SHA_256

```
   K =        00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f
              10 11 12 13 14 15 16 17 18 19 1a 1b 1c 1d 1e 1f

   MAC_KEY =  00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f

   ENC_KEY =  10 11 12 13 14 15 16 17 18 19 1a 1b 1c 1d 1e 1f

   P =        41 20 63 69 70 68 65 72 20 73 79 73 74 65 6d 20
              6d 75 73 74 20 6e 6f 74 20 62 65 20 72 65 71 75
              69 72 65 64 20 74 6f 20 62 65 20 73 65 63 72 65
              74 2c 20 61 6e 64 20 69 74 20 6d 75 73 74 20 62
              65 20 61 62 6c 65 20 74 6f 20 66 61 6c 6c 20 69
              6e 74 6f 20 74 68 65 20 68 61 6e 64 73 20 6f 66
              20 74 68 65 20 65 6e 65 6d 79 20 77 69 74 68 6f
              75 74 20 69 6e 63 6f 6e 76 65 6e 69 65 6e 63 65

   IV =       1a f3 8c 2d c2 b9 6f fd d8 66 94 09 23 41 bc 04

   A =        54 68 65 20 73 65 63 6f 6e 64 20 70 72 69 6e 63
              69 70 6c 65 20 6f 66 20 41 75 67 75 73 74 65 20
              4b 65 72 63 6b 68 6f 66 66 73

   AL =       00 00 00 00 00 00 01 50

   E =        c8 0e df a3 2d df 39 d5 ef 00 c0 b4 68 83 42 79
              a2 e4 6a 1b 80 49 f7 92 f7 6b fe 54 b9 03 a9 c9
              a9 4a c9 b4 7a d2 65 5c 5f 10 f9 ae f7 14 27 e2
              fc 6f 9b 3f 39 9a 22 14 89 f1 63 62 c7 03 23 36
              09 d4 5a c6 98 64 e3 32 1c f8 29 35 ac 40 96 c8
              6e 13 33 14 c5 40 19 e8 ca 79 80 df a4 b9 cf 1b
              38 4c 48 6f 3a 54 c5 10 78 15 8e e5 d7 9d e5 9f
              bd 34 d8 48 b3 d6 95 50 a6 76 46 34 44 27 ad e5

   M =        65 2c 3f a3 6b 0a 7c 5b 32 19 fa b3 a3 0b c1 c4
              e6 e5 45 82 47 65 15 f0 ad 9f 75 a2 b7 1c 73 ef

   T =        65 2c 3f a3 6b 0a 7c 5b 32 19 fa b3 a3 0b c1 c4
```

C.2. Test Cases for AES_256_CBC_HMAC_SHA_512

```
K =         00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f
            10 11 12 13 14 15 16 17 18 19 1a 1b 1c 1d 1e 1f
            20 21 22 23 24 25 26 27 28 29 2a 2b 2c 2d 2e 2f
            30 31 32 33 34 35 36 37 38 39 3a 3b 3c 3d 3e 3f

MAC_KEY =   00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f
            10 11 12 13 14 15 16 17 18 19 1a 1b 1c 1d 1e 1f

ENC_KEY =   20 21 22 23 24 25 26 27 28 29 2a 2b 2c 2d 2e 2f
            30 31 32 33 34 35 36 37 38 39 3a 3b 3c 3d 3e 3f

P =         41 20 63 69 70 68 65 72 20 73 79 73 74 65 6d 20
            6d 75 73 74 20 6e 6f 74 20 62 65 20 72 65 71 75
            69 72 65 64 20 74 6f 20 62 65 20 73 65 63 72 65
            74 2c 20 61 6e 64 20 69 74 20 6d 75 73 74 20 62
            65 20 61 62 6c 65 20 74 6f 20 66 61 6c 6c 20 69
            6e 74 6f 20 74 68 65 20 68 61 6e 64 73 20 6f 66
            20 74 68 65 20 65 6e 65 6d 79 20 77 69 74 68 6f
            75 74 20 69 6e 63 6f 6e 76 65 6e 69 65 6e 63 65

IV =        1a f3 8c 2d c2 b9 6f fd d8 66 94 09 23 41 bc 04

A =         54 68 65 20 73 65 63 6f 6e 64 20 70 72 69 6e 63
            69 70 6c 65 20 6f 66 20 41 75 67 75 73 74 65 20
            4b 65 72 63 6b 68 6f 66 66 73

AL =        00 00 00 00 00 00 01 50

E =         4a ff aa ad b7 8c 31 c5 da 4b 1b 59 0d 10 ff bd
            3d d8 d5 d3 02 42 35 26 91 2d a0 37 ec bc c7 bd
            82 2c 30 1d d6 7c 37 3b cc b5 84 ad 3e 92 79 c2
            e6 d1 2a 13 74 b7 7f 07 75 53 df 82 94 10 44 6b
            36 eb d9 70 66 29 6a e6 42 7e a7 5c 2e 08 46 a1
            1a 09 cc f5 37 0d c8 0b fe cb ad 28 c7 3f 09 b3
            a3 b7 5e 66 2a 25 94 41 0a e4 96 b2 e2 e6 60 9e
            31 e6 e0 2c c8 37 f0 53 d2 1f 37 ff 4f 51 95 0b
            be 26 38 d0 9d d7 a4 93 09 30 80 6d 07 03 b1 f6

M =         4d d3 b4 c0 88 a7 f4 5c 21 68 39 64 5b 20 12 bf
            2e 62 69 a8 c5 6a 81 6d bc 1b 26 77 61 95 5b c5
            fd 30 a5 65 c6 16 ff b2 f3 64 ba ec e6 8f c4 07
            53 bc fc 02 5d de 36 93 75 4a a1 f5 c3 37 3b 9c

T =         4d d3 b4 c0 88 a7 f4 5c 21 68 39 64 5b 20 12 bf
            2e 62 69 a8 c5 6a 81 6d bc 1b 26 77 61 95 5b c5
```

Appendix D. Acknowledgements

   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.

   The Authenticated Encryption with AES-CBC and HMAC-SHA
   [I-D.mcgrew-aead-aes-cbc-hmac-sha2] specification, upon which the
   AES_CBC_HMAC_SHA2 algorithms are based, was written by David A.
   McGrew and Kenny Paterson. The test cases for AES_CBC_HMAC_SHA2 are
   based upon those for [I-D.mcgrew-aead-aes-cbc-hmac-sha2] by John
   Foley.

   This specification is the work of the JOSE Working Group, which
   includes dozens of active and dedicated participants. In particular,
   the following individuals contributed ideas, feedback, and wording
   that influenced this specification:

   Dirk Balfanz, Richard Barnes, John Bradley, Brian Campbell, Breno de
   Medeiros, Yaron Y. Goland, Dick Hardt, Jeff Hodges, Edmund Jay, James
   Manger, Tony Nadalin, Axel Nennker, John Panzer, Emmanuel Raviart,
   Nat Sakimura, Jim Schaad, Hannes Tschofenig, and Sean Turner.

   Jim Schaad and Karen O'Donoghue chaired the JOSE working group and
   Sean Turner and Stephen Farrell served as Security area directors
   during the creation of this specification.

Appendix E. Document History

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

   -12

   o In the Direct Key Agreement case, the Concat KDF AlgorithmID is
      set to the octets of the UTF-8 representation of the "enc" header
      parameter value.

   o Restored the "apv" (agreement PartyVInfo) parameter.

   o Moved the "epk", "apu", and "apv" Header Parameter definitions to
      be with the algorithm descriptions that use them.

   o Changed terminology from "block encryption" to "content
      encryption".

   -11

   o Removed the Encrypted Key value from the AAD computation since it
      is already effectively integrity protected by the encryption
      process. The AAD value now only contains the representation of
      the JWE Encrypted Header.

   o Removed "apv" (agreement PartyVInfo) since it is no longer used.

   o Added more information about the use of PartyUInfo during key
      agreement.

   o Use the keydatalen as the SuppPubInfo value for the Concat KDF
      when doing key agreement, as RFC 2631 does.

   o Added algorithm identifiers for RSASSA-PSS with SHA-256 and SHA-
      512.

   o Added a Parameter Information Class value to the JSON Web Key
      Parameters registry, which registers whether the parameter conveys
      public or private information.

   -10

   o Changed the JWE processing rules for multiple recipients so that a
      single AAD value contains the header parameters and encrypted key
      values for all the recipients, enabling AES GCM to be safely used
      for multiple recipients.

   -09

   o Expanded the scope of the JWK parameters to include private and
      symmetric key representations, as specified by
      draft-jones-jose-json-private-and-symmetric-key-00.

   o Changed term "JWS Secured Input" to "JWS Signing Input".

   o Changed from using the term "byte" to "octet" when referring to 8
      bit values.

   o Specified that AES Key Wrap uses the default initial value
      specified in Section 2.2.3.1 of RFC 3394. This addressed issue
      #19.

   o Added Key Management Mode definitions to terminology section and
      used the defined terms to provide clearer key management
      instructions. This addressed issue #5.

o Replaced "A128CBC+HS256" and "A256CBC+HS512" with "A128CBC-HS256"
   and "A256CBC-HS512". The new algorithms perform the same
   cryptographic computations as [I-D.mcgrew-aead-aes-cbc-hmac-sha2],
   but with the Initialization Vector and Authentication Tag values
   remaining separate from the Ciphertext value in the output
   representation. Also deleted the header parameters "epu"
   (encryption PartyUInfo) and "epv" (encryption PartyVInfo), since
   they are no longer used.

o Changed from using the term "Integrity Value" to "Authentication
   Tag".

-08

o Changed the name of the JWK key type parameter from "alg" to
   "kty".

o Replaced uses of the term "AEAD" with "Authenticated Encryption",
   since the term AEAD in the RFC 5116 sense implied the use of a
   particular data representation, rather than just referring to the
   class of algorithms that perform authenticated encryption with
   associated data.

o Applied editorial improvements suggested by Jeff Hodges. Many of
   these simplified the terminology used.

o Added seriesInfo information to Internet Draft references.

-07

o Added a data length prefix to PartyUInfo and PartyVInfo values.

o Changed the name of the JWK RSA modulus parameter from "mod" to
   "n" and the name of the JWK RSA exponent parameter from "xpo" to
   "e", so that the identifiers are the same as those used in RFC
   3447.

o Made several local editorial changes to clean up loose ends left
   over from to the decision to only support block encryption methods
   providing integrity.

-06

o Removed the "int" and "kdf" parameters and defined the new
   composite Authenticated Encryption algorithms "A128CBC+HS256" and
   "A256CBC+HS512" to replace the former uses of AES CBC, which
   required the use of separate integrity and key derivation
   functions.

o Included additional values in the Concat KDF calculation -- the
   desired output size and the algorithm value, and optionally
   PartyUInfo and PartyVInfo values. Added the optional header
   parameters "apu" (agreement PartyUInfo), "apv" (agreement
   PartyVInfo), "epu" (encryption PartyUInfo), and "epv" (encryption
   PartyVInfo).

o Changed the name of the JWK RSA exponent parameter from "exp" to
   "xpo" so as to allow the potential use of the name "exp" for a
   future extension that might define an expiration parameter for
   keys. (The "exp" name is already used for this purpose in the JWT
   specification.)

o Applied changes made by the RFC Editor to RFC 6749's registry
   language to this specification.

-05

o 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.

o Updated open issues.

-04

o Added text requiring that any leading zero bytes be retained in
   base64url encoded key value representations for fixed-length
   values.

o 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."

o Described additional open issues.

o Applied editorial suggestions.

-03

o Always use a 128 bit "authentication tag" size for AES GCM,
   regardless of the key size.

o Specified that use of a 128 bit IV is REQUIRED with AES CBC. It
   was previously RECOMMENDED.

o Removed key size language for ECDSA algorithms, since the key size
   is implied by the algorithm being used.

o 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.

o 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.

o Clarified that the "mod" and "exp" values are unsigned.

o Added Implementation Requirements columns to algorithm tables and
   Implementation Requirements entries to algorithm registries.

o Changed AES Key Wrap to RECOMMENDED.

o Moved registries JSON Web Signature and Encryption Header
   Parameters and JSON Web Signature and Encryption Type Values to
   the JWS specification.

o Moved JSON Web Key Parameters registry to the JWK specification.

o Changed registration requirements from RFC Required to
   Specification Required with Expert Review.

o Added Registration Template sections for defined registries.

o Added Registry Contents sections to populate registry values.

o 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".

o Added "Collision Resistant Namespace" to the terminology section.

o Numerous editorial improvements.

-02

o 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
   Authentication Tag.

o Defined minimum required key sizes for algorithms without
   specified key sizes.

o Defined KDF output key sizes.

o Specified the use of PKCS #5 padding with AES CBC.

o Generalized text to allow key agreement to be employed as an
   alternative to key wrapping or key encryption.

o Clarified that ECDH-ES is a key agreement algorithm.

o Required implementation of AES-128-KW and AES-256-KW.

o Removed the use of "A128GCM" and "A256GCM" for key wrapping.

o Removed "A512KW" since it turns out that it's not a standard
   algorithm.

o Clarified the relationship between "typ" header parameter values
   and MIME types.

o 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.

o 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.

o Moved algorithm-specific definitions from JWK to JWA.

o Reformatted to give each member definition its own section
   heading.

-01

o Moved definition of "alg":"none" for JWSs here from the JWT
   specification since this functionality is likely to be useful in
   more contexts that just for JWTs.

o Added Advanced Encryption Standard (AES) Key Wrap Algorithm using
   512 bit keys ("A512KW").

o 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".

   o Corrected the Magic Signatures reference.

   o Made other editorial improvements suggested by JOSE working group
      participants.

   -00

   o Created the initial IETF draft based upon
      draft-jones-json-web-signature-04 and
      draft-jones-json-web-encryption-02 with no normative changes.

   o Changed terminology to no longer call both digital signatures and
      HMACs "signatures".

Author's Address

   Michael B. Jones
   Microsoft

   Email: mbj@microsoft.com
   URI: http://self-issued.info/