JOSE Working Group	M. Jones
Internet-Draft	Microsoft
Intended status: Standards Track	December 27, 2012
Expires: June 30, 2013	

TOC

JSON Web Algorithms (JWA)
draft-ietf-jose-json-web-algorithms-08

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

TOC

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

TOC

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

---

2. Terminology

TOC

2.1. Terms Incorporated from the JWS Specification

TOC

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

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

JSON Text Object
: A UTF-8 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 that describes the digital signature or MAC operation applied to create the JWS Signature value.

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

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

Base64url Encoding
: The URL- and filename-safe Base64 encoding described in **RFC 4648** [RFC4648], Section 5, with the (non URL-safe) '=' padding characters omitted, as permitted by Section 3.2. (See Appendix C of **[JWS]** for notes on implementing base64url encoding without padding.)

Encoded JWS Header
: Base64url encoding of the JWS Header.

Encoded JWS Payload
: Base64url encoding of the JWS Payload.

Encoded JWS Signature
: Base64url encoding of the JWS Signature.

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

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

2.2. Terms Incorporated from the JWE Specification

TOC

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

JSON Web Encryption (JWE)
: A data structure representing an encrypted message. The structure consists of five parts: the JWE Header, the JWE Encrypted Key, the JWE Initialization Vector, the JWE Ciphertext, and the JWE Integrity Value.

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

Ciphertext
: An encrypted representation of the Plaintext.

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

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

Content Master Key (CMK)
: A key from which the CEK and CIK are derived. When key wrapping or key encryption are employed, the CMK is randomly generated and encrypted to the recipient as the JWE Encrypted Key. When direct encryption with a shared symmetric key is employed, the CMK is the shared key. When key agreement without key wrapping is employed, the CMK is the result of the key agreement algorithm.

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

JWE Header
: A JSON Text Object that describes the encryption operations applied to create the JWE Encrypted Key, the JWE Ciphertext, and the JWE Integrity Value.

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

JWE Initialization Vector
: A byte array containing the Initialization Vector used when encrypting the Plaintext.

JWE Ciphertext
: A byte array containing the Ciphertext.

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

Encoded JWE Header
: Base64url encoding of the JWE 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 Integrity Value
: Base64url encoding of the JWE Integrity Value.

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.

TOC

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 data structure that represents a public key.

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

TOC

2.4. Defined Terms

These terms are defined for use by this specification:

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

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

TOC

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.

TOC

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

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

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

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

TOC

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 Secured Input, which therefore demonstrates that whoever generated the MAC was in possession of the secret. The means of exchanging the shared key is outside the scope of this specification.

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

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

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

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

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

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

TOC

3.3. Digital Signature with RSA SHA-256, RSA SHA-384, or RSA 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 RSA digital signature using the respective hash function.

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

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

1. Generate a digital signature of the bytes of the ASCII representation of the JWS Secured Input using RSASSA-PKCS1-V1_5-SIGN and the SHA-256 hash function with the desired private key. The output will be a byte array.
2. Base64url encode the resulting byte array.

The output is the Encoded JWS Signature for that JWS.

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

1. Take the Encoded JWS Signature and base64url decode it into a byte array. If decoding fails, the JWS MUST be rejected.
2. Submit the bytes of the ASCII representation of the JWS Secured Input and the public key corresponding to the private key used by the signer to the RSASSA-

PKCS1-V1_5-VERIFY algorithm using SHA-256 as the hash function.
3. If the validation fails, the JWS MUST be rejected.

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

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

TOC

The Elliptic Curve Digital Signature Algorithm (ECDSA) **[DSS]** provides for the use of Elliptic Curve cryptography, which is able to provide equivalent security to RSA cryptography but using shorter key sizes and with greater processing speed. This means that ECDSA digital signatures will be substantially smaller in terms of length than equivalently strong RSA digital signatures.

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

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

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

The output is the Encoded JWS Signature for the JWS.

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

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

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

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

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

3.5. Using the Algorithm "none"

TOC

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

3.6. Additional Digital Signature/MAC Algorithms and Parameters

TOC

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

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

4. Cryptographic Algorithms for JWE

TOC

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

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

TOC

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

alg Parameter Value	Key Encryption or Agreement Algorithm	Implementation Requirements
RSA1_5	RSAES-PKCS1-V1_5 **[RFC3447]**	REQUIRED
RSA-OAEP	RSAES using Optimal Asymmetric Encryption Padding (OAEP) **[RFC3447]**, with the default parameters specified by RFC 3447 in Section A.2.1	OPTIONAL
A128KW	Advanced Encryption Standard (AES) Key Wrap Algorithm **[RFC3394]** using 128 bit keys	RECOMMENDED
A256KW	AES Key Wrap Algorithm using 256 bit keys	RECOMMENDED
dir	Direct use of a shared symmetric key as the Content Master Key (CMK) for the block encryption step (rather than using the symmetric key to wrap the CMK)	RECOMMENDED
ECDH-ES	Elliptic Curve Diffie-Hellman Ephemeral Static **[RFC6090]** key agreement using the Concat KDF, as defined in Section 5.8.1 of **[NIST.800-56A]**, with the agreed-upon key being used directly as the Content Master Key (CMK) (rather than being used to wrap the CMK), as specified in **Section 4.7**	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 Master Key (CMK) with the A128KW function (rather than being used directly as the CMK)	RECOMMENDED
ECDH-ES+A256KW	Elliptic Curve Diffie-Hellman Ephemeral Static key agreement per ECDH-ES and **Section 4.7**, but where the agreed-upon key is used to wrap the Content Master Key (CMK) with the A256KW function (rather than being used directly as the CMK)	RECOMMENDED

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

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

TOC

The table below is the set of enc (encryption method) header parameter values that are defined by this specification for use with JWE. These algorithms are used to encrypt the Plaintext, which produces the Ciphertext.

enc Parameter Value	Block Encryption Algorithm	Implementation Requirements
A128CBC+HS256	Composite Authenticated Encryption algorithm using Advanced Encryption Standard (AES) in Cipher Block Chaining (CBC) mode with PKCS #5 padding **[AES] [NIST.800-38A]** with an integrity calculation using HMAC SHA-256, using a 256 bit CMK (and 128 bit CEK) as specified in **Section 4.8**	REQUIRED
A256CBC+HS512	Composite Authenticated Encryption algorithm using AES in CBC mode with PKCS #5 padding with an integrity calculation using HMAC SHA-512, using a 512 bit CMK (and 256 bit CEK) as specified in **Section 4.8**	REQUIRED
A128GCM	AES in Galois/Counter Mode (GCM) **[AES] [NIST.800-38D]** using 128 bit keys	RECOMMENDED
A256GCM	AES GCM using 256 bit keys	RECOMMENDED

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

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

4.3. Key Encryption with RSAES-PKCS1-V1_5

TOC

This section defines the specifics of encrypting a JWE CMK with RSAES-PKCS1-V1_5 **[RFC3447]**. The alg header parameter value RSA1_5 is used in this case.

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

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

4.4. Key Encryption with RSAES OAEP

TOC

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

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

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

TOC

4.5. Key Encryption with AES Key Wrap

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

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

TOC

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 Master Key (CMK) value for the `enc` algorithm. An empty byte array is used as the JWE Encrypted Key value. The `alg` header parameter value `dir` is used in this case.

TOC

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 Master Key (CMK) for the `enc` algorithm, or (2) as a symmetric key used to wrap the CMK with either the `A128KW` or `A256KW` algorithms. The `alg` header parameter values `ECDH-ES`, `ECDH-ES+A128KW`, and `ECDH-ES+A256KW` are respectively used in this case.

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

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

TOC

4.7.1. Key Derivation for "ECDH-ES"

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 a byte array.

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
: This is set to the concatenation of keydatalen represented as a 32 bit big endian integer and the bytes 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 bytes, and Datalen is a fixed-length, big endian 32 bit counter that indicates the length (in bytes) 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 bytes in Data. Otherwise, Datalen is set to 0 and Data is set to the empty byte string.

PartyVInfo
: The PartyVInfo value is of the form Datalen || Data, where Data is a variable-length string of zero or more bytes, and Datalen is a fixed-length, big endian 32 bit counter that indicates the length (in bytes) 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 bytes in Data. Otherwise, Datalen is set to 0 and Data is set to the empty byte string.

SuppPubInfo
: This is set to the empty byte string.

SuppPrivInfo
: This is set to the empty byte string.

4.8. Composite Plaintext Encryption Algorithms "A128CBC+HS256" and "A256CBC+HS512"

TOC

This section defines two composite `enc` algorithms that perform plaintext encryption using non-Authenticated Encryption algorithms and add an integrity check calculation, so that the resulting composite algorithms perform Authenticated Encryption. These composite algorithms derive a Content Encryption Key (CEK) and a Content Integrity Key (CIK) from a Content Master Key, per **Section 4.8.1**. They perform block encryption with AES CBC, per **Section 4.8.2**. Finally, they add an integrity check using HMAC SHA-2 algorithms of matching strength, per **Section 4.8.3**.

A 256 bit Content Master Key (CMK) value is used with the `A128CBC+HS256` algorithm. A 512 bit Content Master Key (CMK) value is used with the `A256CBC+HS512` algorithm.

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

4.8.1. Key Derivation for "A128CBC+HS256" and "A256CBC+HS512"

TOC

The key derivation process derives CEK and CIK values from the CMK. This section defines the specifics of deriving keys for the `enc` algorithms `A128CBC+HS256` and `A256CBC+HS512`.

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 or SHA-512, respectively. The Concat KDF parameters are set as follows:

Z
: This is set to the Content Master Key (CMK).

keydatalen
: This is set to the number of bits in the desired output key.

AlgorithmID
: This is set to the concatenation of keydatalen represented as a 32 bit big endian integer and the bytes of the UTF-8 representation of the `enc` header parameter value.

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

PartyVInfo
: The PartyVInfo value is of the form Datalen || Data, where Data is a variable-length

string of zero or more bytes, and Datalen is a fixed-length, big endian 32 bit counter that indicates the length (in bytes) of Data, with || being concatenation. If an epv (encryption PartyVInfo) header parameter is present, Data is set to the result of base64url decoding the epv value and Datalen is set to the number of bytes in Data. Otherwise, Datalen is set to 0 and Data is set to the empty byte string.

SuppPubInfo
: This is set to the bytes of one of the ASCII strings "Encryption" ([69, 110, 99, 114, 121, 112, 116, 105, 111, 110]) or "Integrity" ([73, 110, 116, 101, 103, 114, 105, 116, 121]) respectively, depending upon whether the CEK or CIK is being generated.

SuppPrivInfo
: This is set to the empty byte string.

To compute the CEK from the CMK, the ASCII label "Encryption" is used for the SuppPubInfo value. For `A128CBC+HS256`, the keydatalen is 128 and the digest function used is SHA-256. For `A256CBC+HS512`, the keydatalen is 256 and the digest function used is SHA-512.

To compute the CIK from the CMK, the ASCII label "Integrity" is used for the SuppPubInfo value. For `A128CBC+HS256`, the keydatalen is 256 and the digest function used is SHA-256. For `A256CBC+HS512`, the keydatalen is 512 and the digest function used is SHA-512.

Example derivation computations are shown in Appendices A.4 and A.5 of **[JWE]**.

4.8.2. Encryption Calculation for "A128CBC+HS256" and "A256CBC+HS512"

TOC

This section defines the specifics of encrypting the JWE Plaintext with Advanced Encryption Standard (AES) in Cipher Block Chaining (CBC) mode with PKCS #5 padding **[AES]** **[NIST.800-38A]** using 128 or 256 bit keys. The `enc` header parameter values `A128CBC+HS256` or `A256CBC+HS512` are respectively used in this case.

The CEK is used as the encryption key.

Use of an initialization vector of size 128 bits is REQUIRED with these algorithms.

4.8.3. Integrity Calculation for "A128CBC+HS256" and "A256CBC+HS512"

TOC

This section defines the specifics of computing the JWE Integrity Value for the `enc` algorithms `A128CBC+HS256` and `A256CBC+HS512`. This value is computed as a MAC of the JWE parameters to be secured.

The MAC input value is the bytes of the ASCII representation of the concatenation of the Encoded JWE Header, a period ('.') character, the Encoded JWE Encrypted Key, a second period character ('.'), the Encoded JWE Initialization Vector, a third period ('.') character, and the Encoded JWE Ciphertext. (Equivalently, this input value is the concatenation of the "additional authenticated data" value, a byte containing an ASCII period character, and the bytes of the ASCII representation of the Encoded JWE Ciphertext.)

The CIK is used as the MAC key.

For `A128CBC+HS256`, HMAC SHA-256 is used as the MAC algorithm. For `A256CBC+HS512`, HMAC SHA-512 is used as the MAC algorithm.

The resulting MAC value is used as the JWE Integrity Value. (Equivalently, this value is the "authentication tag" output for the algorithm.) The same integrity calculation is performed during decryption. During decryption, the computed integrity value must match the received JWE Integrity Value.

4.9. Plaintext Encryption with AES GCM

TOC

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

The CMK is used as the encryption key.

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

The "additional authenticated data" parameter is used to secure the header and key values. (The "additional authenticated data" value used is the bytes of the ASCII representation of the concatenation of the Encoded JWE Header, a period ('.') character, the Encoded JWE Encrypted Key, a second period character ('.'), and the Encoded JWE Initialization Vector, per Section 5 of the JWE specification.) This same "additional authenticated data" value is used when decrypting as well.

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

The JWE Integrity Value is set to be the "authentication tag" value produced by the encryption. During decryption, the received JWE Integrity Value is used as the "authentication tag" value.

Examples using this algorithm are shown in Appendices A.1 and A.3 of **[JWE]**.

TOC

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 MAY be used to determine whether the object is a JWS or JWE.

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

TOC

5. Cryptographic Algorithms for JWK

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

TOC

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

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

TOC

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

5.2.1. "crv" (Curve) Parameter

TOC

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

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

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

5.2.2. "x" (X Coordinate) Parameter

TOC

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

5.2.3. "y" (Y Coordinate) Parameter

TOC

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

5.3. JWK Parameters for RSA Keys

TOC

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

5.3.1. "n" (Modulus) Parameter

TOC

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 a byte array. The array representation MUST not be shortened to omit any leading zero bytes. For instance, when representing 2048 bit integers, the byte array to be base64url encoded MUST contain 256 bytes, including any leading zero bytes.

5.3.2. "e" (Exponent) Parameter

TOC

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 a byte array. The array representation MUST utilize the minimum number of bytes to represent the value. For instance, when representing the value 65537, the byte array to be base64url encoded MUST consist of the three bytes [1, 0, 1].

5.4. Additional Key Types and Parameters

TOC

Public keys using additional key types MAY 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

TOC

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

TOC

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. Registration Template

TOC

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 may be followed by a "+" or "-". The use of "+" indicates that the requirement strength is likely to be increased in a future version of the specification. The use of "-" indicates that the requirement strength is likely to be decreased in a future version of the specification.

Change Controller:
: For Standards Track RFCs, state "IETF". For others, give the name of the responsible party. Other details (e.g., postal address, 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

TOC

- Algorithm Name: `HS256`
- Algorithm Usage Location(s): `alg`
- Implementation Requirements: REQUIRED
- Change Controller: IETF
- Specification Document(s): **Section 3.1** of [[this document]]

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

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

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

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

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

- Algorithm Name: `ES256`
- Algorithm Usage Location(s): `alg`

- Implementation Requirements: RECOMMENDED+
- Change Controller: IETF
- Specification Document(s): **Section 3.1** of [[this document]]

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

- Algorithm Name: `ES512`
- Algorithm Usage Location(s): `alg`
- Implementation Requirements: OPTIONAL
- Change Controller: IETF
- Specification Document(s): **Section 3.1** of [[this document]]

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

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

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

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

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

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

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

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

- Algorithm Name: `ECDH-ES+A256KW`
- Algorithm Usage Location(s): `alg`
- Implementation Requirements: RECOMMENDED
- Change Controller: IETF

- Specification Document(s): **Section 4.1** of [[this document]]

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

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

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

- Algorithm Name: `A256GCM`
- Algorithm Usage Location(s): `enc`
- Implementation Requirements: RECOMMENDED
- Change Controller: IETF
- Specification Document(s): **Section 4.2** of [[this document]]

TOC

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**.

TOC

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 may be followed by a "+" or "-". The use of "+" indicates that the requirement strength is likely to be increased in a future version of the specification. The use of "-" indicates that the requirement strength is likely to be decreased in a future version of the specification.

Specification Document(s):
: Reference to the document(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.

TOC

6.2.2. Initial Registry Contents

- "kty" Parameter Value: `EC`
- Implementation Requirements: RECOMMENDED+
- Change Controller: IETF

- Specification Document(s): **Section 5.1** of [[this document]]

- "kty" Parameter Value: `RSA`
- Implementation Requirements: REQUIRED
- Change Controller: IETF
- Specification Document(s): **Section 5.1** of [[this document]]

6.3. JSON Web Key Parameters Registration

TOC

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

6.3.1. Registry Contents

TOC

- Parameter Name: `crv`
- Change Controller: IETF
- Specification Document(s): **Section 5.2.1** of [[this document]]

- Parameter Name: `x`
- Change Controller: IETF
- Specification Document(s): **Section 5.2.2** of [[this document]]

- Parameter Name: `y`
- Change Controller: IETF
- Specification Document(s): **Section 5.2.3** of [[this document]]

- Parameter Name: `n`
- Change Controller: IETF
- Specification Document(s): **Section 5.3.1** of [[this document]]

- Parameter Name: `e`
- Change Controller: IETF
- Specification Document(s): **Section 5.3.2** of [[this document]]

7. Security Considerations

TOC

All of the security issues faced by any cryptographic application must be faced by a JWS/JWE/JWK agent. Among these issues are protecting the user's private 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 concerns are listed here.

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

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

Algorithms of matching strength should be used together whenever possible. For instance, when AES Key Wrap is used with a given key size, using the same key size is recommended when AES GCM is also used.

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

Keys used with RSAES-PKCS1-v1_5 must follow the constraints in Section 7.2 of RFC 3447 **[RFC3447]**. In particular, keys with a low public key exponent value must not be used.

Plaintext JWSs (JWSs that use the `alg` value `none`) provide no integrity protection. Thus, they must only be used in contexts where the payload is secured by means other than a digital signature or MAC value, or need not be secured.

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

TOC

8. References

TOC

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), December 2012 (HTML).

[JWK] Jones, M., "JSON Web Key (JWK)," draft-ietf-jose-json-web-key (work in progress), December 2012 (HTML).

[JWS] Jones, M., Bradley, J., and N. Sakimura, "JSON Web Signature (JWS)," draft-ietf-jose-json-web-signature (work in progress), December 2012 (HTML).

[NIST.800-38A] National Institute of Standards and Technology (NIST), "Recommendation for Block Cipher Modes of Operation," NIST PUB 800-38A, December 2001.

[NIST.800-38D] National Institute of Standards and Technology (NIST), "Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC," NIST PUB 800-38D, December 2001.

[NIST.800-56A] National Institute of Standards and Technology (NIST), "Recommendation for Pair-Wise Key Establishment Schemes Using Discrete Logarithm Cryptography (Revised)," NIST PUB 800-56A, March 2007.

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

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

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

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

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

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

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

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

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

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

[SHS] National Institute of Standards and Technology, "Secure Hash Standard (SHS)," FIPS PUB 180-3, October 2008.

[USASCII] American National Standards Institute, "Coded Character Set -- 7-bit American Standard Code for Information Interchange," ANSI X3.4, 1986.

TOC

8.2. Informative References

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

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

[JCA] Oracle, "Java Cryptography Architecture," 2011.

[JSE] Bradley, J. and N. Sakimura (editor), "JSON Simple Encryption," September 2010.

[JSS] Bradley, J. and N. Sakimura (editor), "JSON Simple Sign," September 2010.

[MagicSignatures] Panzer (editor), J., Laurie, B., and D. Balfanz, "Magic Signatures," January 2011.

[RFC3275]	Eastlake, D., Reagle, J., and D. Solo, "**(Extensible Markup Language) XML-Signature Syntax and Processing**," RFC 3275, March 2002 (**TXT**).
[RFC4122]	**Leach, P.**, **Mealling, M.**, and **R. Salz**, "**A Universally Unique IDentifier (UUID) URN Namespace**," RFC 4122, July 2005 (**TXT**, **HTML**, **XML**).
[W3C.CR-xmldsig-core2-20120124]	Yiu, K., Solo, D., Eastlake, D., Datta, P., Hirsch, F., Reagle, J., Cantor, S., and T. Roessler, "**XML Signature Syntax and Processing Version 2.0**," World Wide Web Consortium CR CR-xmldsig-core2-20120124, January 2012 (**HTML**).
[W3C.CR-xmlenc-core1-20120313]	Eastlake, D., Reagle, J., Roessler, T., and F. Hirsch, "**XML Encryption Syntax and Processing Version 1.1**," World Wide Web Consortium CR CR-xmlenc-core1-20120313, March 2012 (**HTML**).
[W3C.REC-xmlenc-core-20021210]	Eastlake, D. and J. Reagle, "**XML Encryption Syntax and Processing**," World Wide Web Consortium Recommendation REC-xmlenc-core-20021210, December 2002 (**HTML**).

TOC

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 using SHA-256 hash algorithm	RS256	http://www.w3.org/2001/04/xmldsig-more#rsa-sha256	SHA256withRSA	1.2.840.113549.1.1.11
RSASSA using SHA-384 hash algorithm	RS384	http://www.w3.org/2001/04/xmldsig-more#rsa-sha384	SHA384withRSA	1.2.840.113549.1.1.12
RSASSA using SHA-512 hash 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	ES512	http://www.w3.org/2001/04/xmldsig-more#ecdsa-sha512	SHA512withECDSA	1.2.840.10045.4.3.4

TOC

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.

Algorithm	JWE	XML ENC	JCA
RSAES-PKCS1-V1_5	RSA1_5	http://www.w3.org/2001/04/xmlenc#rsa-1_5	RSA/ECB/PKCS1Padding
RSAES using Optimal Asymmetric Encryption Padding (OAEP)	RSA-OAEP	http://www.w3.org/2001/04/xmlenc#rsa-oaep-mgf1p	RSA/ECB/OAEPWithSHA-1AndMGF1Padding
Elliptic Curve Diffie-Hellman Ephemeral Static	ECDH-ES	http://www.w3.org/2009/xmlenc11#ECDH-ES	
Advanced Encryption Standard (AES) Key Wrap Algorithm using 128 bit keys	A128KW	http://www.w3.org/2001/04/xmlenc#kw-aes128	
AES Key Wrap Algorithm using 256 bit keys	A256KW	http://www.w3.org/2001/04/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

---

TOC

Appendix C. 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.

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.

TOC

Appendix D. Open Issues

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

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

- No known open issues.

TOC

Appendix E. Document History

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

-08

- Changed the name of the JWK key type parameter from `alg` to `kty`.
- 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.
- Applied editorial improvements suggested by Jeff Hodges. Many of these simplified the terminology used.
- Added seriesInfo information to Internet Draft references.

-07

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

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

-05

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

-04

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

-03

- Always use a 128 bit "authentication tag" size for AES GCM, regardless of the key size.
- Specified that use of a 128 bit IV is REQUIRED with AES CBC. It was previously RECOMMENDED.
- Removed key size language for ECDSA algorithms, since the key size is implied by the algorithm being used.
- Stated that the `int` key size must be the same as the hash output size (and not larger, as was previously allowed) so that its size is defined for key generation purposes.
- Added the `kdf` (key derivation function) header parameter to provide crypto agility for key derivation. The default KDF remains the Concat KDF with the SHA-256 digest function.
- Clarified that the `mod` and `exp` values are unsigned.
- Added Implementation Requirements columns to algorithm tables and Implementation Requirements entries to algorithm registries.
- Changed AES Key Wrap to RECOMMENDED.
- Moved registries JSON Web Signature and Encryption Header Parameters and JSON Web Signature and Encryption Type Values to the JWS specification.
- Moved JSON Web Key Parameters registry to the JWK specification.
- Changed registration requirements from RFC Required to Specification Required with Expert Review.
- Added Registration Template sections for defined registries.
- Added Registry Contents sections to populate registry values.
- No longer say "the UTF-8 representation of the JWS Secured Input (which is the same as the ASCII representation)". Just call it "the ASCII representation of the JWS Secured Input".
- Added "Collision Resistant Namespace" to the terminology section.
- Numerous editorial improvements.

-02

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

JSON Web Signature and Encryption Algorithms, JSON Web Signature and Encryption "typ" Values, JSON Web Key Parameters, and JSON Web Key Algorithm Families.

- Moved algorithm-specific definitions from JWK to JWA.
- Reformatted to give each member definition its own section heading.

-01

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

-00

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

TOC

Author's Address

Michael B. Jones
Microsoft
Email: mbj@microsoft.com
URI: http://self-issued.info/