JOSE Working Group
Internet-Draft
Intended status: Standards Track
Expires: July 24, 2014

M. Jones
Microsoft

January 20, 2014

JSON Web Algorithms (JWA)
draft-ietf-jose-json-web-algorithms-20

Abstract

The JSON Web Algorithms (JWA) specification registers cryptographic algorithms and identifiers to be used with the JSON Web Signature (JWS), JSON Web Encryption (JWE), and JSON Web Key (JWK) specifications. It defines several IANA registries for these identifiers.

Status of this Memo

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

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

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

This Internet-Draft will expire on July 24, 2014.

Table of Contents

1. Introduction

The JSON Web Algorithms (JWA) specification registers 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. It defines several IANA registries for these identifiers. All these specifications utilize JavaScript Object Notation (JSON) [I-D.ietf-json-rfc4627bis] based data structures. This specification also describes the semantics and operations that are specific to these algorithms and key types.

Registering the algorithms and identifiers here, 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. This also allows changes to the JWS, JWE, and JWK specifications without changing this document.

Names defined by this specification are short because a core goal is for the resulting representations to be compact.

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]. If these words are used without being spelled in uppercase then they are to be interpreted with their normal natural language meanings.

BASE64URL(OCTETS) denotes the base64url encoding of OCTETS, per Section 2.

UTF8(STRING) denotes the octets of the UTF-8 [RFC3629] representation of STRING.

ASCII(STRING) denotes the octets of the ASCII [USASCII] representation of STRING.

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

2. Terminology

These terms defined by the JSON Web Signature (JWS) [JWS] specification are incorporated into this specification: "JSON Web Signature (JWS)", "JWS Header", "JWS Payload", "JWS Signature", "JWS Protected Header", "Base64url Encoding", and "JWS Signing Input".

These terms defined by the JSON Web Encryption (JWE) [JWE] specification are incorporated into this specification: "JSON Web Encryption (JWE)", "Authenticated Encryption", "Plaintext", "Ciphertext", "Additional Authenticated Data (AAD)", "Authentication Tag", "Content Encryption Key (CEK)", "JWE Header", "JWE Encrypted Key", "JWE Initialization Vector", "JWE Ciphertext", "JWE Authentication Tag", "JWE Protected Header", "Key Management Mode", "Key Encryption", "Key Wrapping", "Direct Key Agreement", "Key Agreement with Key Wrapping", and "Direct Encryption".

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

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.

3. Cryptographic Algorithms for Digital Signatures and MACs

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.

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	Required
HS384	HMAC using SHA-384	Optional
HS512	HMAC using SHA-512	Optional
RS256	RSASSA-PKCS-v1_5 using SHA-256	Recommended
RS384	RSASSA-PKCS-v1_5 using SHA-384	Optional
RS512	RSASSA-PKCS-v1_5 using SHA-512	Optional
ES256	ECDSA using P-256 and SHA-256	Recommended+

ES384	ECDSA using P-384 and SHA-384	Optional
ES512	ECDSA using P-521 and SHA-512	Optional
PS256	RSASSA-PSS using SHA-256 and MGF1 with SHA-256	Optional
PS384	RSASSA-PSS using SHA-384 and MGF1 with SHA-384	Optional
PS512	RSASSA-PSS using SHA-512 and MGF1 with SHA-512	Optional
none	No digital signature or MAC performed	Optional

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.1 for a table cross-referencing the JWS digital signature and MAC "alg" (algorithm) values defined in this specification with the equivalent identifiers used by other standards and software packages.

3.2. HMAC with SHA-2 Functions

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 whoever generated the MAC was in possession of the MAC key. The algorithm for implementing and validating HMACs is provided in RFC 2104 [RFC2104].

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 JWS Signing Input as the "text" value, and using the shared key. The HMAC output value is the JWS Signature.

The following "alg" (algorithm) Header Parameter values are used to indicate that the JWS Signature is an HMAC value computed using the corresponding algorithm:

alg Parameter Value	MAC Algorithm
HS256	HMAC using SHA-256
HS384	HMAC using SHA-384
HS512	HMAC using SHA-512

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 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 value. Alternatively, the computed HMAC value can be base64url encoded and compared to the received encoded JWS Signature value, as this comparison produces the same result as comparing the unencoded values. In either case, if the values match, the HMAC has been validated.

Securing content and validation 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 algorithms 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

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-2 [SHS] hash functions.

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: Generate a digital signature of the JWS Signing Input using RSASSA-PKCS1-V1_5-SIGN and the SHA-256 hash function with the desired private key. This is the JWS Signature value.

The following "alg" (algorithm) Header Parameter values are used to indicate that the JWS Signature is a digital signature value computed using the corresponding algorithm:

alg Parameter Value	Digital Signature Algorithm
RS256	RSASSA-PKCS-v1_5 using SHA-256
RS384	RSASSA-PKCS-v1_5 using SHA-384
RS512	RSASSA-PKCS-v1_5 using SHA-512

The RSASSA-PKCS1-V1_5 SHA-256 digital signature for a JWS is validated as follows: Submit the JWS Signing Input, the JWS Signature, 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.

Signing and validation 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 algorithms instead of SHA-256.

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

3.4. Digital Signature with ECDSA

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

1. Generate a digital signature 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 octet sequence 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. The resulting 64 octet sequence is the JWS Signature value.

The following "alg" (algorithm) Header Parameter values are used to indicate that the JWS Signature is a digital signature value computed using the corresponding algorithm:

alg Parameter Value	Digital Signature Algorithm
ES256	ECDSA using P-256 and SHA-256
ES384	ECDSA using P-384 and SHA-384
ES512	ECDSA using P-521 and SHA-512

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

1. The JWS Signature value MUST be a 64 octet sequence. If it is not a 64 octet sequence, the validation has failed.

2. 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).

3. Submit the JWS Signing Input R, S and the public key (x, y) to the ECDSA P-256 SHA-256 validator.

Signing and validation 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 algorithms 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

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 and SHA-2 hash functions, always using the same hash function for both the RSASSA-PSS hash function and the

MGF1 hash function. The size of the salt value is the same size as the hash function output. All other algorithm parameters use the defaults specified in Section A.2.3 of RFC 3447.

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: Generate a digital signature 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. This is the JWS signature value.

The following "alg" (algorithm) Header Parameter values are used to indicate that the JWS Signature is a digital signature value computed using the corresponding algorithm:

alg Parameter Value	Digital Signature Algorithm
PS256	RSASSA-PSS using SHA-256 and MGF1 with SHA-256
PS384	RSASSA-PSS using SHA-384 and MGF1 with SHA-384
PS512	RSASSA-PSS using SHA-512 and MGF1 with SHA-512

The RSASSA-PSS SHA-256 digital signature for a JWS is validated as follows: Submit the JWS Signing Input, the JWS Signature, 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.

Signing and validation with the RSASSA-PSS SHA-384 and RSASSA-PSS SHA-512 algorithms 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". A Plaintext JWS MUST use the "alg" value "none", and is formatted identically to other JWSs, but MUST use the empty octet sequence as its JWS Signature value. Receivers MUST verify that the JWS Signature value is the empty octet sequence. See Section 8.5 for security considerations associated with using this algorithm.

4. Cryptographic Algorithms for Key Management

JWE uses cryptographic algorithms to encrypt or determine the Content Encryption Key (CEK).

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	(none)	Required
RSA-OAEP	RSAES using OAEP with default parameters	(none)	Optional
A128KW	AES Key Wrap with default initial value using 128 bit key	(none)	Recommended
A192KW	AES Key Wrap with default initial value using 192 bit key	(none)	Optional
A256KW	AES Key Wrap with default initial value using 256 bit key	(none)	Recommended
dir	Direct use of a shared symmetric key as the CEK	(none)	Recommended
ECDH-ES	Elliptic Curve Diffie-Hellman Ephemeral Static key agreement using Concat KDF	"epk", "apu", "apv"	Recommended+

ECDH-ES+A128KW	ECDH-ES using Concat KDF and CEK wrapped with "A128KW"	"epk", "apu", "apv"	Recommended
ECDH-ES+A192KW	ECDH-ES using Concat KDF and CEK wrapped with "A192KW"	"epk", "apu", "apv"	Optional
ECDH-ES+A256KW	ECDH-ES using Concat KDF and CEK wrapped with "A256KW"	"epk", "apu", "apv"	Recommended
A128GCMKW	Key wrapping with AES GCM using 128 bit key	"iv", "tag"	Optional
A192GCMKW	Key wrapping with AES GCM using 192 bit key	"iv", "tag"	Optional
A256GCMKW	Key wrapping with AES GCM using 256 bit key	"iv", "tag"	Optional
PBES2-HS256+A128KW	PBES2 with HMAC SHA-256 and "A128KW" wrapping	"p2s", "p2c"	Optional
PBES2-HS384+A192KW	PBES2 with HMAC SHA-384 and "A192KW" wrapping	"p2s", "p2c"	Optional
PBES2-HS512+A256KW	PBES2 with HMAC SHA-512 and "A256KW" wrapping	"p2s", "p2c"	Optional

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.

See Appendix A.2 for a table cross-referencing the JWE "alg"

(algorithm) values defined in this specification with the equivalent identifiers used by other standards and software packages.

4.2. 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 for this algorithm.

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].

4.3. 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. (Those default parameters are using a hash function of SHA-1 and a mask generation function of MGF1 with SHA-1.) The "alg" Header Parameter value "RSA-OAEP" is used for this algorithm.

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].

4.4. 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.

The following "alg" (algorithm) Header Parameter values are used to indicate that the JWE Encrypted Key is the result of encrypting the CEK using the corresponding algorithm and key size:

alg Parameter Value	Key Management Algorithm
A128KW	AES Key Wrap with default initial value using 128 bit key
A192KW	AES Key Wrap with default initial value using 192 bit key
A256KW	AES Key Wrap with default initial value using 256 bit key

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

4.5. 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.

Refer to the security considerations on key lifetimes in Section 8.2 and AES GCM in Section 8.4 when considering utilizing direct encryption.

4.6. 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], in combination with 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 the "A128KW", "A192KW", or "A256KW" algorithms, in the Key Agreement with Key Wrapping mode.

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

In Direct Key Agreement mode, 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. The "alg" Header Parameter value "ECDH-ES" is used in the Direct Key Agreement mode.

In Key Agreement with Key Wrapping mode, the output of the Concat KDF MUST be a key of the length needed for the specified key wrapping algorithm. In this case, the JWE Encrypted Key is the CEK wrapped with the agreed upon key.

The following "alg" (algorithm) Header Parameter values are used to indicate that the JWE Encrypted Key is the result of encrypting the CEK using the result of the key agreement algorithm as the key encryption key for the corresponding key wrapping algorithm:

alg Parameter Value	Key Management Algorithm
ECDH-ES+A128KW	ECDH-ES using Concat KDF and CEK wrapped with "A128KW"
ECDH-ES+A192KW	ECDH-ES using Concat KDF and CEK wrapped with "A192KW"
ECDH-ES+A256KW	ECDH-ES using Concat KDF and CEK wrapped with "A256KW"

4.6.1. Header Parameters Used for ECDH Key Agreement

The following Header Parameter names are used for key agreement as defined below.

4.6.1.1. "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] public key value. It MUST contain only public key parameters and SHOULD contain only the minimum JWK parameters necessary to represent the key; other JWK parameters included can be checked for consistency and honored or can be ignored. This Header Parameter MUST be present and MUST be understood and processed by implementations when these algorithms are used.

4.6.1.2. "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.

4.6.1.3. "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.

4.6.2. 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", "ECDH-ES+A192KW", and "ECDH-ES+A256KW", this is 128, 192, and 256, respectively.

AlgorithmID The AlgorithmID 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. In the Direct Key Agreement case, Data is set to the octets of the UTF-8 representation of the "enc" Header Parameter value. In the Key Agreement with Key Wrapping case, Data 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. 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. 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.

Applications need to specify how the "apu" and "apv" parameters are used for that application. The "apu" and "apv" values MUST be distinct, when used. Applications wishing to conform to [NIST.800-56A] need to provide values that meet the requirements of that document, e.g., by using values that identify the sender and recipient. Alternatively, applications MAY conduct key derivation in a manner similar to The Diffie-Hellman Key Agreement Method [RFC2631]: In that case, the "apu" field MAY either be omitted or represent a random 512-bit value (analogous to PartyAInfo in Ephemeral-Static mode in [RFC2631]) and the "apv" field should not be present.

See Appendix C for an example key agreement computation using this method.

4.7. Key Encryption with AES GCM

This section defines the specifics of encrypting a JWE Content Encryption Key (CEK) with Advanced Encryption Standard (AES) in Galois/Counter Mode (GCM) [AES] [NIST.800-38D].

Use of an Initialization Vector of size 96 bits is REQUIRED with this algorithm. The Initialization Vector is represented in base64url encoded form as the "iv" (initialization vector) Header Parameter value.

The Additional Authenticated Data value used is the empty octet string.

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

The JWE Encrypted Key value is the Ciphertext output.

The Authentication Tag output is represented in base64url encoded form as the "tag" (authentication tag) Header Parameter value.

The following "alg" (algorithm) Header Parameter values are used to indicate that the JWE Encrypted Key is the result of encrypting the CEK using the corresponding algorithm and key size:

alg Parameter Value	Key Management Algorithm
A128GCMKW	Key wrapping with AES GCM using 128 bit key
A192GCMKW	Key wrapping with AES GCM using 192 bit key
A256GCMKW	Key wrapping with AES GCM using 256 bit key

4.7.1. Header Parameters Used for AES GCM Key Encryption

The following Header Parameters are used for AES GCM key encryption.

4.7.1.1. "iv" (Initialization Vector) Header Parameter

The "iv" (initialization vector) Header Parameter value is the base64url encoded representation of the Initialization Vector value used for the key encryption operation. This Header Parameter MUST be present and MUST be understood and processed by implementations when these algorithms are used.

4.7.1.2. "tag" (Authentication Tag) Header Parameter

The "tag" (authentication tag) Header Parameter value is the base64url encoded representation of the Authentication Tag value resulting from the key encryption operation. This Header Parameter MUST be present and MUST be understood and processed by implementations when these algorithms are used.

4.8. Key Encryption with PBES2

This section defines the specifies of performing password-based encryption of a JWE CEK, by first deriving a key encryption key from a user-supplied password using PBES2 schemes as specified in Section 6.2 of [RFC2898], then by encrypting the JWE CEK using the derived key.

These algorithms use HMAC SHA-2 algorithms as the Pseudo-Random Function (PRF) for the PBKDF2 key derivation and AES Key Wrap [RFC3394] for the encryption scheme. The PBES2 password input is an octet sequence; if the password to be used is represented as a text string rather than an octet sequence, the UTF-8 encoding of the text string MUST be used as the octet sequence. The salt MUST be provided as the "p2s" Header Parameter value, and MUST be base64url decoded to obtain the value. The iteration count parameter MUST be provided as the "p2c" Header Parameter value. The algorithms respectively use HMAC SHA-256, HMAC SHA-384, and HMAC SHA-512 as the PRF and use 128, 192, and 256 bit AES Key Wrap keys. Their derived-key lengths respectively are 16, 24, and 32 octets.

The following "alg" (algorithm) Header Parameter values are used to indicate that the JWE Encrypted Key is the result of encrypting the CEK using the result of the corresponding password-based encryption algorithm as the key encryption key for the corresponding key wrapping algorithm:

alg Parameter Value	Key Management Algorithm
PBES2-HS256+A128KW	PBES2 with HMAC SHA-256 and "A128KW" wrapping
PBES2-HS384+A192KW	PBES2 with HMAC SHA-384 and "A192KW" wrapping
PBES2-HS512+A256KW	PBES2 with HMAC SHA-512 and "A256KW" wrapping

See Appendix C of JSON Web Key (JWK) [JWK] for an example key encryption computation using "PBES2-HS256+A128KW".

4.8.1. Header Parameters Used for PBES2 Key Encryption

The following Header Parameters are used for Key Encryption with PBES2.

4.8.1.1. "p2s" (PBES2 salt) Parameter

The "p2s" (PBES2 salt) Header Parameter contains the PBKDF2 salt value, encoded using base64url. This Header Parameter MUST be present and MUST be understood and processed by implementations when these algorithms are used.

The salt expands the possible keys that can be derived from a given password. A salt value containing 8 or more octets MUST be used. A new salt value MUST be generated randomly for every encryption operation; see [RFC4086] for considerations on generating random values.

4.8.1.2. "p2c" (PBES2 count) Parameter

The "p2c" (PBES2 count) Header Parameter contains the PBKDF2 iteration count, represented as a positive integer. This Header Parameter MUST be present and MUST be understood and processed by implementations when these algorithms are used.

The iteration count adds computational expense, ideally compounded by the possible range of keys introduced by the salt. A minimum iteration count of 1000 is RECOMMENDED.

5. Cryptographic Algorithms for Content Encryption

JWE uses cryptographic algorithms to encrypt the Plaintext.

5.1. "enc" (Encryption Algorithm) Header Parameter Values for JWE

The table below is the set of "enc" (encryption algorithm) 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	AES_128_CBC_HMAC_SHA_256 authenticated encryption algorithm, as defined in Section 5.2.3	(none)	Required
A192CBC-HS384	AES_192_CBC_HMAC_SHA_384 authenticated encryption algorithm, as defined in Section 5.2.4	(none)	Optional
A256CBC-HS512	AES_256_CBC_HMAC_SHA_512 authenticated encryption algorithm, as defined in Section 5.2.5	(none)	Required
A128GCM	AES GCM using 128 bit key	(none)	Recommended
A192GCM	AES GCM using 192 bit key	(none)	Optional
A256GCM	AES GCM using 256 bit key	(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 produce JWE Ciphertext and JWE Authentication Tag values.

See Appendix A.3 for a table cross-referencing the JWE "enc" (encryption algorithm) values defined in this specification with the equivalent identifiers used by other standards and software packages.

5.2. 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 three instances of this family, the first using 128 bit CBC keys and HMAC SHA-256, the second using 192 bit CBC keys and HMAC SHA-384, and the third using 256 bit CBC keys and HMAC SHA-512. Test cases for these algorithms can be found in Appendix B.

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 option is discussed in Appendix B of that specification. 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.

5.2.1. 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.

5.2.2. 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 5.2.2.1 and Section 5.2.2.2 define the generic encryption and decryption algorithms. Section 5.2.3 and Section 5.2.5 define instances of AES_CBC_HMAC_SHA2 that specify those details.

5.2.2.1. 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 5.2.3 and Section 5.2.5). 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.

5.2.2.2. 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 5.2.2.1.

2. The integrity and authenticity of A and E are checked by computing an HMAC with the inputs as in Step 5 of Section 5.2.2.1. 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.

5.2.3. 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 and a 128 bit initialization vector (IV) value.

The AES_CBC_HMAC_SHA2 parameters specific to AES_128_CBC_HMAC_SHA_256 are:

The input key K is 32 octets long.

ENC_KEY_LEN is 16 octets.

MAC_KEY_LEN is 16 octets.

The SHA-256 hash algorithm is used for the HMAC.

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

5.2.4. AES_192_CBC_HMAC_SHA_384

AES_192_CBC_HMAC_SHA_384 is based on AES_128_CBC_HMAC_SHA_256, but with the following differences:

The input key K is 48 octets long instead of 32.

ENC_KEY_LEN is 24 octets instead of 16.

MAC_KEY_LEN is 24 octets instead of 16.

SHA-384 is used for the HMAC instead of SHA-256.

The HMAC SHA-384 value is truncated to T_LEN=24 octets instead of 16.

5.2.5. 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:

The input key K is 64 octets long instead of 32.

ENC_KEY_LEN is 32 octets instead of 16.

MAC_KEY_LEN is 32 octets instead of 16.

SHA-512 is used for the HMAC instead of SHA-256.

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

5.2.6. Content Encryption with AES_CBC_HMAC_SHA2

The following "enc" (encryption algorithm) Header Parameter values are used to indicate that the JWE Ciphertext and JWE Authentication Tag values have been computed using the corresponding algorithm:

enc Parameter Value	Content Encryption Algorithm
A128CBC-HS256	AES_128_CBC_HMAC_SHA_256 authenticated encryption algorithm, as defined in Section 5.2.3
A192CBC-HS384	AES_192_CBC_HMAC_SHA_384 authenticated encryption algorithm, as defined in Section 5.2.4
A256CBC-HS512	AES_256_CBC_HMAC_SHA_512 authenticated encryption algorithm, as defined in Section 5.2.5

5.3. Content 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]. The "enc" Header Parameter values "A128GCM", "A192GCM", 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 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.

The following "enc" (encryption algorithm) Header Parameter values are used to indicate that the JWE Ciphertext and JWE Authentication

Tag values have been computed using the corresponding algorithm and key size:

enc Parameter Value	Content Encryption Algorithm
A128GCM	AES GCM using 128 bit key
A192GCM	AES GCM using 192 bit key
A256GCM	AES GCM using 256 bit key

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

6. Cryptographic Algorithms for Keys

A JSON Web Key (JWK) [JWK] is a JSON data structure that represents a cryptographic key. These keys can be either asymmetric or symmetric. They can hold both public and private information about the key. This section defines the parameters for keys using the algorithms specified by this document.

6.1. "kty" (Key Type) Parameter Values

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]	Recommended+
RSA	RSA [RFC3447]	Required
oct	Octet sequence (used to represent symmetric keys)	Required

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

6.2. Parameters for Elliptic Curve Keys

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

6.2.1. Parameters for Elliptic Curve Public Keys

An elliptic curve public key is represented by a pair of coordinates drawn from a finite field, which together define a point on an elliptic curve. The following members MUST be present for elliptic curve public keys:

- "crv"
- "x"
- "y"

SEC1 [SEC1] point compression is not supported for any values.

6.2.1.1. "crv" (Curve) Parameter

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

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

These values are registered in the IANA JSON Web Key Elliptic Curve registry defined in Section 7.6. 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.

6.2.1.2. "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 octet string representation of the coordinate, as defined in Section 2.3.5 of SEC1 [SEC1]. The length of this octet string MUST be the full size of a coordinate for the curve specified in the "crv" parameter. For example, if the value of "crv" is "P-521", the octet string must be 66 octets long.

6.2.1.3. "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 octet string representation of the coordinate, as defined in Section 2.3.5 of SEC1 [SEC1]. The length of this octet string MUST be the full size of a coordinate for the curve specified in the "crv"

parameter. For example, if the value of "crv" is "P-521", the octet string must be 66 octets long.

6.2.2. 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.

6.2.2.1. "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 octet string representation of the private key value, as defined in Sections C.4 and 2.3.7 of SEC1 [SEC1]. The length of this octet string MUST be ceiling(log-base-2(n)/8) octets (where n is the order of the curve).

6.3. Parameters for RSA Keys

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

6.3.1. Parameters for RSA Public Keys

The following members MUST be present for RSA public keys.

6.3.1.1. "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 octet sequence MUST utilize the minimum number of octets to represent the value.

6.3.1.2. "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 octet sequence 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].

6.3.2. 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. The parameter "d" is REQUIRED for RSA private keys. The others enable optimizations and SHOULD be included by producers of JWKs representing RSA private keys. If the producer includes any of the other private key parameters, then all of the others MUST be present, with the exception of "oth", which MUST only be present when more than two prime factors were used. The consumer of a JWK MAY choose to accept an RSA private key that does not contain a complete set of the private key parameters other than "d", including JWKs in which "d" is the only RSA private key parameter included.

6.3.2.1. "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 octet sequence MUST utilize the minimum number of octets to represent the value.

6.3.2.2. "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. The octet sequence MUST utilize the minimum number of octets to represent the value.

6.3.2.3. "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. The octet sequence MUST utilize the minimum number of octets to represent the value.

6.3.2.4. "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. The octet sequence MUST utilize the minimum number of octets to represent the value.

6.3.2.5. "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. The octet sequence MUST utilize the minimum number of octets to represent the value.

6.3.2.6. "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. The octet sequence MUST utilize the minimum number of octets to represent the value.

6.3.2.7. "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:

6.3.2.7.1. "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. The octet sequence MUST utilize the minimum number of octets to represent the value.

6.3.2.7.2. "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. The octet sequence MUST utilize the minimum number of octets to represent the value.

6.3.2.7.3. "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. The octet sequence MUST utilize the minimum number of octets to represent the value.

6.4. Parameters for Symmetric Keys

When the JWK "kty" member value is "oct" (octet sequence), the member "k" is used to represent a symmetric key (or another key whose value is a single octet sequence). An "alg" member SHOULD also be present to identify the algorithm intended to be used with the key, unless the application uses another means or convention to determine the algorithm used.

6.4.1. "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.

7. 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 the 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. Registration requests that are undetermined for a period

longer than 21 days can be brought to the IESG's attention (using the iesg@iesg.org mailing list) for resolution.

Criteria that should be applied by the Designated Expert(s) includes determining whether the proposed registration duplicates existing functionality, determining whether it is likely to be of general applicability or whether it is useful only for a single application, and whether the registration makes sense.

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

It is suggested that multiple Designated Experts be appointed who are able to represent the perspectives of different applications using this specification, in order to enable broadly-informed review of registration decisions. In cases where a registration decision could be perceived as creating a conflict of interest for a particular Expert, that Expert should defer to the judgment of the other Expert(s).

7.1. 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 algorithm) Header Parameters. The registry records the algorithm name, the algorithm usage locations, implementation requirements, and a reference to the specification that defines it. The same algorithm name can be registered multiple times, provided that the sets of usage locations are disjoint.

It is suggested that when algorithms can use keys of different lengths, that the length of the key be included in the algorithm name. This allows readers of the JSON text to easily make security consideration decisions.

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. Changes of implementation requirements are only permitted on a Specification Required basis, with the new specification defining the revised implementation requirements level.

7.1.1. Registration Template

Algorithm Name:
The name requested (e.g., "example"). This name is case-sensitive. Names may not match other registered names in a case-insensitive manner unless the Designated Expert(s) state that there is a compelling reason to allow an exception in this particular case.

Algorithm Description:
Brief description of the Algorithm (e.g., "Example description").

Algorithm Usage Location(s):
The algorithm usage location. This must be one or more of the values "alg" or "enc" if the algorithm is to be used with JWS or JWE. The value "JWK" is used if the algorithm identifier will be used as a JWK "alg" member value, but will not be used with JWS or JWE; this could be the case, for instance, for non-authenticated encryption algorithms. Other values may be used with the approval of a Designated Expert.

JOSE Implementation Requirements:
The algorithm implementation requirements for JWS and JWE, which must be one the words Required, Recommended, Optional, Deprecated, or Prohibited. 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. Any identifiers registered for non-authenticated encryption algorithms or other algorithms that are otherwise unsuitable for direct use as JWS or JWE algorithms must be registered as "Prohibited".

Change Controller:
For Standards Track RFCs, state "IESG". 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.

7.1.2. Initial Registry Contents

- Algorithm Name: "HS256"
- Algorithm Description: HMAC using SHA-256

- Algorithm Usage Location(s): "alg"
- JOSE Implementation Requirements: Required
- Change Controller: IESG
- Specification Document(s): Section 3.1 of [[this document]]

- Algorithm Name: "HS384"
- Algorithm Description: HMAC using SHA-384
- Algorithm Usage Location(s): "alg"
- JOSE Implementation Requirements: Optional
- Change Controller: IESG
- Specification Document(s): Section 3.1 of [[this document]]

- Algorithm Name: "HS512"
- Algorithm Description: HMAC using SHA-512
- Algorithm Usage Location(s): "alg"
- JOSE Implementation Requirements: Optional
- Change Controller: IESG
- Specification Document(s): Section 3.1 of [[this document]]

- Algorithm Name: "RS256"
- Algorithm Description: RSASSA-PKCS-v1_5 using SHA-256
- Algorithm Usage Location(s): "alg"
- JOSE Implementation Requirements: Recommended
- Change Controller: IESG
- Specification Document(s): Section 3.1 of [[this document]]

- Algorithm Name: "RS384"
- Algorithm Description: RSASSA-PKCS-v1_5 using SHA-384
- Algorithm Usage Location(s): "alg"
- JOSE Implementation Requirements: Optional
- Change Controller: IESG
- Specification Document(s): Section 3.1 of [[this document]]

- Algorithm Name: "RS512"
- Algorithm Description: RSASSA-PKCS-v1_5 using SHA-512
- Algorithm Usage Location(s): "alg"
- JOSE Implementation Requirements: Optional
- Change Controller: IESG
- Specification Document(s): Section 3.1 of [[this document]]

- Algorithm Name: "ES256"
- Algorithm Description: ECDSA using P-256 and SHA-256
- Algorithm Usage Location(s): "alg"
- JOSE Implementation Requirements: Recommended+
- Change Controller: IESG
- Specification Document(s): Section 3.1 of [[this document]]

- Algorithm Name: "ES384"
- Algorithm Description: ECDSA using P-384 and SHA-384
- Algorithm Usage Location(s): "alg"
- JOSE Implementation Requirements: Optional
- Change Controller: IESG
- Specification Document(s): Section 3.1 of [[this document]]

- Algorithm Name: "ES512"
- Algorithm Description: ECDSA using P-521 and SHA-512
- Algorithm Usage Location(s): "alg"
- JOSE Implementation Requirements: Optional
- Change Controller: IESG
- Specification Document(s): Section 3.1 of [[this document]]

- Algorithm Name: "PS256"
- Algorithm Description: RSASSA-PSS using SHA-256 and MGF1 with SHA-256
- Algorithm Usage Location(s): "alg"
- JOSE Implementation Requirements: Optional
- Change Controller: IESG
- Specification Document(s): Section 3.1 of [[this document]]

- Algorithm Name: "PS384"
- Algorithm Description: RSASSA-PSS using SHA-384 and MGF1 with SHA-384
- Algorithm Usage Location(s): "alg"
- JOSE Implementation Requirements: Optional
- Change Controller: IESG
- Specification Document(s): Section 3.1 of [[this document]]

- Algorithm Name: "PS512"
- Algorithm Description: RSASSA-PSS using SHA-512 and MGF1 with SHA-512
- Algorithm Usage Location(s): "alg"
- JOSE Implementation Requirements: Optional
- Change Controller: IESG
- Specification Document(s): Section 3.1 of [[this document]]

- Algorithm Name: "none"
- Algorithm Description: No digital signature or MAC performed
- Algorithm Usage Location(s): "alg"
- JOSE Implementation Requirements: Optional
- Change Controller: IESG
- Specification Document(s): Section 3.1 of [[this document]]

- Algorithm Name: "RSA1_5"

- Algorithm Description: RSAES-PKCS1-V1_5
- Algorithm Usage Location(s): "alg"
- JOSE Implementation Requirements: Required
- Change Controller: IESG
- Specification Document(s): Section 4.1 of [[this document]]

- Algorithm Name: "RSA-OAEP"
- Algorithm Description: RSAES using OAEP with default parameters
- Algorithm Usage Location(s): "alg"
- JOSE Implementation Requirements: Optional
- Change Controller: IESG
- Specification Document(s): Section 4.1 of [[this document]]

- Algorithm Name: "A128KW"
- Algorithm Description: AES Key Wrap using 128 bit key
- Algorithm Usage Location(s): "alg"
- JOSE Implementation Requirements: Recommended
- Change Controller: IESG
- Specification Document(s): Section 4.1 of [[this document]]

- Algorithm Name: "A192KW"
- Algorithm Description: AES Key Wrap using 192 bit key
- Algorithm Usage Location(s): "alg"
- JOSE Implementation Requirements: Optional
- Change Controller: IESG
- Specification Document(s): Section 4.1 of [[this document]]

- Algorithm Name: "A256KW"
- Algorithm Description: AES Key Wrap using 256 bit key
- Algorithm Usage Location(s): "alg"
- JOSE Implementation Requirements: Recommended
- Change Controller: IESG
- Specification Document(s): Section 4.1 of [[this document]]

- Algorithm Name: "dir"
- Algorithm Description: Direct use of a shared symmetric key
- Algorithm Usage Location(s): "alg"
- JOSE Implementation Requirements: Recommended
- Change Controller: IESG
- Specification Document(s): Section 4.1 of [[this document]]

- Algorithm Name: "ECDH-ES"
- Algorithm Description: ECDH-ES using Concat KDF
- Algorithm Usage Location(s): "alg"
- JOSE Implementation Requirements: Recommended+
- Change Controller: IESG

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

- Algorithm Name: "ECDH-ES+A128KW"
- Algorithm Description: ECDH-ES using Concat KDF and "A128KW" wrapping
- Algorithm Usage Location(s): "alg"
- JOSE Implementation Requirements: Recommended
- Change Controller: IESG
- Specification Document(s): Section 4.1 of [[this document]]

- Algorithm Name: "ECDH-ES+A192KW"
- Algorithm Description: ECDH-ES using Concat KDF and "A192KW" wrapping
- Algorithm Usage Location(s): "alg"
- JOSE Implementation Requirements: Optional
- Change Controller: IESG
- Specification Document(s): Section 4.1 of [[this document]]

- Algorithm Name: "ECDH-ES+A256KW"
- Algorithm Description: ECDH-ES using Concat KDF and "A256KW" wrapping
- Algorithm Usage Location(s): "alg"
- JOSE Implementation Requirements: Recommended
- Change Controller: IESG
- Specification Document(s): Section 4.1 of [[this document]]

- Algorithm Name: "A128GCMKW"
- Algorithm Description: Key wrapping with AES GCM using 128 bit key
- Algorithm Usage Location(s): "alg"
- JOSE Implementation Requirements: Optional
- Change Controller: IESG
- Specification Document(s): Section 4.7 of [[this document]]

- Algorithm Name: "A192GCMKW"
- Algorithm Description: Key wrapping with AES GCM using 192 bit key
- Algorithm Usage Location(s): "alg"
- JOSE Implementation Requirements: Optional
- Change Controller: IESG
- Specification Document(s): Section 4.7 of [[this document]]

- Algorithm Name: "A256GCMKW"
- Algorithm Description: Key wrapping with AES GCM using 256 bit key
- Algorithm Usage Location(s): "alg"
- JOSE Implementation Requirements: Optional
- Change Controller: IESG
- Specification Document(s): Section 4.7 of [[this document]]

- Algorithm Name: "PBES2-HS256+A128KW"
- Algorithm Description: PBES2 with HMAC SHA-256 and "A128KW" wrapping
- Algorithm Usage Location(s): "alg"
- JOSE Implementation Requirements: Optional
- Change Controller: IESG
- Specification Document(s): Section 4.8 of [[this document]]

- Algorithm Name: "PBES2-HS384+A192KW"
- Algorithm Description: PBES2 with HMAC SHA-384 and "A192KW" wrapping
- Algorithm Usage Location(s): "alg"
- JOSE Implementation Requirements: Optional
- Change Controller: IESG
- Specification Document(s): Section 4.8 of [[this document]]

- Algorithm Name: "PBES2-HS512+A256KW"
- Algorithm Description: PBES2 with HMAC SHA-512 and "A256KW" wrapping
- Algorithm Usage Location(s): "alg"
- JOSE Implementation Requirements: Optional
- Change Controller: IESG
- Specification Document(s): Section 4.8 of [[this document]]

- Algorithm Name: "A128CBC-HS256"
- Algorithm Description: AES_128_CBC_HMAC_SHA_256 authenticated encryption algorithm
- Algorithm Usage Location(s): "enc"
- JOSE Implementation Requirements: Required
- Change Controller: IESG
- Specification Document(s): Section 5.1 of [[this document]]

- Algorithm Name: "A192CBC-HS384"
- Algorithm Description: AES_192_CBC_HMAC_SHA_384 authenticated encryption algorithm
- Algorithm Usage Location(s): "enc"
- JOSE Implementation Requirements: Optional
- Change Controller: IESG
- Specification Document(s): Section 5.1 of [[this document]]

- Algorithm Name: "A256CBC-HS512"
- Algorithm Description: AES_256_CBC_HMAC_SHA_512 authenticated encryption algorithm
- Algorithm Usage Location(s): "enc"
- JOSE Implementation Requirements: Required
- Change Controller: IESG

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

- Algorithm Name: "A128GCM"
- Algorithm Description: AES GCM using 128 bit key
- Algorithm Usage Location(s): "enc"
- JOSE Implementation Requirements: Recommended
- Change Controller: IESG
- Specification Document(s): Section 5.1 of [[this document]]

- Algorithm Name: "A192GCM"
- Algorithm Description: AES GCM using 192 bit key
- Algorithm Usage Location(s): "enc"
- JOSE Implementation Requirements: Optional
- Change Controller: IESG
- Specification Document(s): Section 5.1 of [[this document]]

- Algorithm Name: "A256GCM"
- Algorithm Description: AES GCM using 256 bit key
- Algorithm Usage Location(s): "enc"
- JOSE Implementation Requirements: Recommended
- Change Controller: IESG
- Specification Document(s): Section 5.1 of [[this document]]

7.2. JWE Header Parameter Names Registration

This specification registers the Header Parameter names defined in Section 4.6.1, Section 4.7.1, and Section 4.8.1 in the IANA JSON Web Signature and Encryption Header Parameters registry defined in [JWS].

7.2.1. Registry Contents

- Header Parameter Name: "epk"
- Header Parameter Description: Ephemeral Public Key
- Header Parameter Usage Location(s): JWE
- Change Controller: IESG
- Specification Document(s): Section 4.6.1.1 of [[this document]]

- Header Parameter Name: "apu"
- Header Parameter Description: Agreement PartyUInfo
- Header Parameter Usage Location(s): JWE
- Change Controller: IESG
- Specification Document(s): Section 4.6.1.2 of [[this document]]

- Header Parameter Name: "apv"
- Header Parameter Description: Agreement PartyVInfo
- Header Parameter Usage Location(s): JWE

- Change Controller: IESG
- Specification Document(s): Section 4.6.1.3 of [[this document]]

- Header Parameter Name: "iv"
- Header Parameter Description: Initialization Vector
- Header Parameter Usage Location(s): JWE
- Change Controller: IESG
- Specification Document(s): Section 4.7.1.1 of [[this document]]

- Header Parameter Name: "tag"
- Header Parameter Description: Authentication Tag
- Header Parameter Usage Location(s): JWE
- Change Controller: IESG
- Specification Document(s): Section 4.7.1.2 of [[this document]]

- Header Parameter Name: "p2s"
- Header Parameter Description: PBES2 salt
- Header Parameter Usage Location(s): JWE
- Change Controller: IESG
- Specification Document(s): Section 4.8.1.1 of [[this document]]

- Header Parameter Name: "p2c"
- Header Parameter Description: PBES2 count
- Header Parameter Usage Location(s): JWE
- Change Controller: IESG
- Specification Document(s): Section 4.8.1.2 of [[this document]]

7.3. JSON Web Encryption Compression Algorithms Registry

This specification establishes the IANA JSON Web Encryption Compression Algorithms registry for JWE "zip" member values. The registry records the compression algorithm value and a reference to the specification that defines it.

7.3.1. Registration Template

Compression Algorithm Value:
: The name requested (e.g., "example"). Because a core goal of this specification is for the resulting representations to be compact, it is RECOMMENDED that the name be short -- not to exceed 8 characters without a compelling reason to do so. This name is case-sensitive. Names may not match other registered names in a case-insensitive manner unless the Designated Expert(s) state that there is a compelling reason to allow an exception in this particular case.

Compression Algorithm Description:
Brief description of the compression algorithm (e.g., "Example description").

Change Controller:
For Standards Track RFCs, state "IESG". 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.

7.3.2. Initial Registry Contents

- Compression Algorithm Value: "DEF"
- Compression Algorithm Description: DEFLATE
- Change Controller: IESG
- Specification Document(s): JSON Web Encryption (JWE) [JWE]

7.4. 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, implementation requirements, and a reference to the specification that defines it.

The implementation requirements of a key type MAY be changed over time by the Designated Experts(s) as the cryptographic landscape evolves, for instance, to change the status of a key type to Deprecated, or to change the status of a key type from Optional to Recommended+ or Required. Changes of implementation requirements are only permitted on a Specification Required basis, with the new specification defining the revised implementation requirements level.

7.4.1. Registration Template

"kty" Parameter Value:
The name requested (e.g., "example"). Because a core goal of this specification is for the resulting representations to be compact, it is RECOMMENDED that the name be short -- not to exceed 8 characters without a compelling reason to do so. This name is case-sensitive. Names may not match other registered names in a case-insensitive manner unless the Designated Expert(s) state that there is a compelling reason to allow an exception in this particular case.

Key Type Description:
Brief description of the Key Type (e.g., "Example description").

Change Controller:
For Standards Track RFCs, state "IESG". For others, give the name of the responsible party. Other details (e.g., postal address, email address, home page URI) may also be included.

JOSE Implementation Requirements:
The key type implementation requirements for JWS and JWE, which must be one the words Required, Recommended, Optional, Deprecated, or Prohibited. 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.

7.4.2. Initial Registry Contents

This specification registers the values defined in Section 6.1.

- "kty" Parameter Value: "EC"
- Key Type Description: Elliptic Curve
- JOSE Implementation Requirements: Recommended+
- Change Controller: IESG
- Specification Document(s): Section 6.2 of [[this document]]

- "kty" Parameter Value: "RSA"
- Key Type Description: RSA
- JOSE Implementation Requirements: Required
- Change Controller: IESG
- Specification Document(s): Section 6.3 of [[this document]]

- "kty" Parameter Value: "oct"
- Key Type Description: Octet sequence
- JOSE Implementation Requirements: Required
- Change Controller: IESG
- Specification Document(s): Section 6.4 of [[this document]]

7.5. JSON Web Key Parameters Registration

This specification registers the parameter names defined in Sections 6.2, 6.3, and 6.4 in the IANA JSON Web Key Parameters registry defined in [JWK].

7.5.1. Registry Contents

- Parameter Name: "crv"
- Parameter Description: Curve
- Used with "kty" Value(s): "EC"
- Parameter Information Class: Public
- Change Controller: IESG
- Specification Document(s): Section 6.2.1.1 of [[this document]]

- Parameter Name: "x"
- Parameter Description: X Coordinate
- Used with "kty" Value(s): "EC"
- Parameter Information Class: Public
- Change Controller: IESG
- Specification Document(s): Section 6.2.1.2 of [[this document]]

- Parameter Name: "y"
- Parameter Description: Y Coordinate
- Used with "kty" Value(s): "EC"
- Parameter Information Class: Public
- Change Controller: IESG
- Specification Document(s): Section 6.2.1.3 of [[this document]]

- Parameter Name: "d"
- Parameter Description: ECC Private Key
- Used with "kty" Value(s): "EC"
- Parameter Information Class: Private
- Change Controller: IESG
- Specification Document(s): Section 6.2.2.1 of [[this document]]

- Parameter Name: "n"
- Parameter Description: Modulus
- Used with "kty" Value(s): "RSA"
- Parameter Information Class: Public
- Change Controller: IESG
- Specification Document(s): Section 6.3.1.1 of [[this document]]

- Parameter Name: "e"
- Parameter Description: Exponent
- Used with "kty" Value(s): "RSA"

- Parameter Information Class: Public
- Change Controller: IESG
- Specification Document(s): Section 6.3.1.2 of [[this document]]

- Parameter Name: "d"
- Parameter Description: Private Exponent
- Used with "kty" Value(s): "RSA"
- Parameter Information Class: Private
- Change Controller: IESG
- Specification Document(s): Section 6.3.2.1 of [[this document]]

- Parameter Name: "p"
- Parameter Description: First Prime Factor
- Used with "kty" Value(s): "RSA"
- Parameter Information Class: Private
- Change Controller: IESG
- Specification Document(s): Section 6.3.2.2 of [[this document]]

- Parameter Name: "q"
- Parameter Description: Second Prime Factor
- Used with "kty" Value(s): "RSA"
- Parameter Information Class: Private
- Change Controller: IESG
- Specification Document(s): Section 6.3.2.3 of [[this document]]

- Parameter Name: "dp"
- Parameter Description: First Factor CRT Exponent
- Used with "kty" Value(s): "RSA"
- Parameter Information Class: Private
- Change Controller: IESG
- Specification Document(s): Section 6.3.2.4 of [[this document]]

- Parameter Name: "dq"
- Parameter Description: Second Factor CRT Exponent
- Used with "kty" Value(s): "RSA"
- Parameter Information Class: Private
- Change Controller: IESG
- Specification Document(s): Section 6.3.2.5 of [[this document]]

- Parameter Name: "qi"
- Parameter Description: First CRT Coefficient
- Used with "kty" Value(s): "RSA"
- Parameter Information Class: Private
- Change Controller: IESG
- Specification Document(s): Section 6.3.2.6 of [[this document]]

- Parameter Name: "oth"
- Parameter Description: Other Primes Info
- Used with "kty" Value(s): "RSA"
- Parameter Information Class: Private
- Change Controller: IESG
- Specification Document(s): Section 6.3.2.7 of [[this document]]

- Parameter Name: "k"
- Parameter Description: Key Value
- Used with "kty" Value(s): "oct"
- Parameter Information Class: Private
- Change Controller: IESG
- Specification Document(s): Section 6.4.1 of [[this document]]

7.6. JSON Web Key Elliptic Curve Registry

This specification establishes the IANA JSON Web Key Elliptic Curve registry for JWK "crv" member values. The registry records the curve name, implementation requirements, and a reference to the specification that defines it. This specification registers the parameter names defined in Section 6.2.1.1.

The implementation requirements of a curve MAY be changed over time by the Designated Experts(s) as the cryptographic landscape evolves, for instance, to change the status of a curve to Deprecated, or to change the status of a curve from Optional to Recommended+ or Required. Changes of implementation requirements are only permitted on a Specification Required basis, with the new specification defining the revised implementation requirements level.

7.6.1. Registration Template

Curve Name:
: The name requested (e.g., "example"). Because a core goal of this specification is for the resulting representations to be compact, it is RECOMMENDED that the name be short -- not to exceed 8 characters without a compelling reason to do so. This name is case-sensitive. Names may not match other registered names in a case-insensitive manner unless the Designated Expert(s) state that there is a compelling reason to allow an exception in this particular case.

Curve Description:
: Brief description of the curve (e.g., "Example description").

JOSE Implementation Requirements:
The curve implementation requirements for JWS and JWE, which must be one the words Required, Recommended, Optional, Deprecated, or Prohibited. 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 "IESG". 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.

7.6.2. Initial Registry Contents

- Curve Name: "P-256"
- Curve Description: P-256 curve
- JOSE Implementation Requirements: Recommended+
- Change Controller: IESG
- Specification Document(s): Section 6.2.1.1 of [[this document]]

- Curve Name: "P-384"
- Curve Description: P-384 curve
- JOSE Implementation Requirements: Optional
- Change Controller: IESG
- Specification Document(s): Section 6.2.1.1 of [[this document]]

- Curve Name: "P-521"
- Curve Description: P-521 curve
- JOSE Implementation Requirements: Optional
- Change Controller: IESG
- Specification Document(s): Section 6.2.1.1 of [[this document]]

8. 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.

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.

8.1. Algorithms and Key Sizes will be Deprecated

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

8.2. Key Lifetimes

Many algorithms have associated security considerations related to key lifetimes and/or the number of times that a key may be used. Those security considerations continue to apply when using those algorithms with JOSE data structures.

8.3. RSAES-PKCS1-v1_5 Security Considerations

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.

8.4. AES GCM Security Considerations

Keys used with AES GCM must follow the constraints in Section 8.3 of [NIST.800-38D], which states: "The total number of invocations of the authenticated encryption function shall not exceed 2^32, including all IV lengths and all instances of the authenticated encryption function with the given key". In accordance with this rule, AES GCM MUST NOT be used with the same key value more than 2^32 times.

An Initialization Vector value MUST never be used multiple times with the same AES GCM key. One way to prevent this is to store a counter with the key and increment it with every use. The counter can also be used to prevent exceeding the 2^32 limit above.

This security consideration does not apply to the composite AES-CBC HMAC SHA-2 or AES Key Wrap algorithms.

8.5. Plaintext JWS Security Considerations

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.

Implementations that support plaintext JWS objects MUST NOT accept such objects as valid unless the application specifies that it is acceptable for a specific object to not be integrity-protected. Implementations MUST NOT accept plaintext JWS objects by default. For example, the "verify" method of a hypothetical JWS software library might have a Boolean "acceptUnsigned" parameter that indicates "none" is an acceptable "alg" value. As another example, the "verify" method might take a list of algorithms that are acceptable to the application as a parameter and would reject plaintext JWS values if "none" is not in that list.

In order to mitigate downgrade attacks, applications MUST NOT signal acceptance of plaintext JWS objects at a global level, and SHOULD signal acceptance on a per-object basis. For example, suppose an application accepts JWS objects over two channels, (1) HTTP and (2) HTTPS with client authentication. It requires a JWS signature on objects received over HTTP, but accepts plaintext JWS objects over HTTPS. If the application were to globally indicate that "none" is acceptable, then an attacker could provide it with an unsigned object over HTTP and still have that object successfully validate. Instead, the application needs to indicate acceptance of "none" for each object received over HTTPS (e.g., by setting "acceptUnsigned" to "true" for the first hypothetical JWS software library above), but not for each object received over HTTP.

8.6. Differences between Digital Signatures and MACs

While in many cases, MACs and digital signatures can be used for integrity checking, there are some significant differences between the security properties that each of them provides. These need to be taken into consideration when designing protocols and selecting the algorithms to be used in protocols.

Both signatures and MACs provide for integrity checking -- verifying that the message has not been modified since the integrity value was computed. However, MACs provide for origination identification only under specific circumstances. It can normally be assumed that a private key used for a signature is only in the hands of a single entity (although perhaps a distributed entity, in the case of replicated servers); however, a MAC key needs to be in the hands of all the entities that use it for integrity computation and checking. This means that origination can only be determined if a MAC key is known only to two entities and the receiver knows that it did not create the message. MAC validation cannot be used to prove origination to a third party.

8.7. Denial of Service Attacks

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.

8.8. Reusing Key Material when Encrypting Keys

It is NOT RECOMMENDED to reuse the same key material (Key Encryption Key, Content Encryption Key, Initialization Vector, etc.) to encrypt multiple JWK or JWK Set objects, or to encrypt the same JWK or JWK Set object multiple times. One suggestion for preventing re-use is to always generate a new set key material for each encryption operation, based on the considerations noted in this document as well as from [RFC4086].

8.9. Password Considerations

Passwords are vulnerable to a number of attacks. To help mitigate some of these limitations, this document applies principles from [RFC2898] to derive cryptographic keys from user-supplied passwords.

However, the strength of the password still has a significant impact. A high-entropy password has greater resistance to dictionary attacks. [NIST-800-63-1] contains guidelines for estimating password entropy, which can help applications and users generate stronger passwords.

An ideal password is one that is as large as (or larger than) the

derived key length. However, passwords larger than a certain algorithm-specific size are first hashed, which reduces an attacker's effective search space to the length of the hash algorithm. It is RECOMMENDED that a password used for "PBES2-HS256+A128KW" be no shorter than 16 octets and no longer than 128 octets and a password used for "PBES2-HS512+A256KW" be no shorter than 32 octets and no longer than 128 octets long.

Still, care needs to be taken in where and how password-based encryption is used. These algorithms can still be susceptible to dictionary-based attacks if the iteration count is too small; this is of particular concern if these algorithms are used to protect data that an attacker can have indefinite number of attempts to circumvent the protection, such as protected data stored on a file system.

9. Internationalization Considerations

Passwords obtained from users are likely to require preparation and normalization to account for differences of octet sequences generated by different input devices, locales, etc. It is RECOMMENDED that applications to perform the steps outlined in [I-D.melnikov-precis-saslprepbis] to prepare a password supplied directly by a user before performing key derivation and encryption.

10. References

10.1. 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 186-4, July 2013.

[I-D.ietf-json-rfc4627bis] Bray, T., "The JSON Data Interchange Format", draft-ietf-json-rfc4627bis-10 (work in progress), December 2013.

[I-D.melnikov-precis-saslprepbis] Saint-Andre, P. and A. Melnikov, "Preparation and Comparison of Internationalized Strings Representing Simple User Names and Passwords", draft-melnikov-precis-saslprepbis-04 (work in progress), September 2012.

[JWE] Jones, M., Rescorla, E., and J. Hildebrand, "JSON Web Encryption (JWE)", draft-ietf-jose-json-web-encryption (work in progress), January 2014.

[JWK] Jones, M., "JSON Web Key (JWK)", draft-ietf-jose-json-web-key (work in progress), January 2014.

[JWS] Jones, M., Bradley, J., and N. Sakimura, "JSON Web Signature (JWS)", draft-ietf-jose-json-web-signature (work in progress), January 2014.

[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.

[RFC2898] Kaliski, B., "PKCS #5: Password-Based Cryptography Specification Version 2.0", RFC 2898, September 2000.

[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.

[RFC4086] Eastlake, D., Schiller, J., and S. Crocker, "Randomness Requirements for Security", BCP 106, RFC 4086, June 2005.

[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.

[SEC1] Standards for Efficient Cryptography Group, "SEC 1: Elliptic Curve Cryptography", May 2009.

[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.

10.2. Informative References

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

[I-D.mcgrew-aead-aes-cbc-hmac-sha2]
McGrew, D., Foley, J., and K. Paterson, "Authenticated Encryption with AES-CBC and HMAC-SHA", draft-mcgrew-aead-aes-cbc-hmac-sha2-02 (work in progress), July 2013.

[I-D.miller-jose-jwe-protected-jwk]
Miller, M., "Using JavaScript Object Notation (JSON) Web Encryption (JWE) for Protecting JSON Web Key (JWK) Objects", draft-miller-jose-jwe-protected-jwk-02 (work in progress), June 2013.

[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", 2013.

[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.

[NIST-800-63-1] National Institute of Standards and Technology (NIST), "Electronic Authentication Guideline", NIST 800-63-1, December 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.

[W3C.CR-xmldsig-core2-20120124] Cantor, S., Roessler, T., Eastlake, D., Yiu, K., Reagle, J., Solo, D., Datta, P., and F. Hirsch, "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. Algorithm Identifier Cross-Reference

This appendix contains tables cross-referencing the cryptographic algorithm identifier values defined 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], 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.

A.1. Digital Signature/MAC Algorithm Identifier Cross-Reference

This section contains a table cross-referencing the JWS digital signature and MAC "alg" (algorithm) values defined in this specification with the equivalent identifiers used by other standards and software packages.

JWS	XML DSIG	JCA	OID
HS2	http://www.w3.org/2001/04/xml	HmacSHA256	1.2.840.1135
56	dsig-more#hmac-sha256		49.2.9
HS3	http://www.w3.org/2001/04/xml	HmacSHA384	1.2.840.1135
84	dsig-more#hmac-sha384		49.2.10
HS5	http://www.w3.org/2001/04/xml	HmacSHA512	1.2.840.1135
12	dsig-more#hmac-sha512		49.2.11
RS2	http://www.w3.org/2001/04/xml	SHA256withRS	1.2.840.1135
56	dsig-more#rsa-sha256	A	49.1.1.11
RS3	http://www.w3.org/2001/04/xml	SHA384withRS	1.2.840.1135
84	dsig-more#rsa-sha384	A	49.1.1.12
RS5	http://www.w3.org/2001/04/xml	SHA512withRS	1.2.840.1135
12	dsig-more#rsa-sha512	A	49.1.1.13
ES2	http://www.w3.org/2001/04/xml	SHA256withEC	1.2.840.1004
56	dsig-more#ecdsa-sha256	DSA	5.4.3.2
ES3	http://www.w3.org/2001/04/xml	SHA384withEC	1.2.840.1004
84	dsig-more#ecdsa-sha384	DSA	5.4.3.3
ES5	http://www.w3.org/2001/04/xml	SHA512withEC	1.2.840.1004
12	dsig-more#ecdsa-sha512	DSA	5.4.3.4
PS2	http://www.w3.org/2007/05/xml	SHA256withRS	1.2.840.1135
56	dsig-more#sha256-rsa-MGF1	AandMGF1	49.1.1.10
PS3	http://www.w3.org/2007/05/xml	SHA384withRS	1.2.840.1135
84	dsig-more#sha384-rsa-MGF1	AandMGF1	49.1.1.10
PS5	http://www.w3.org/2007/05/xml	SHA512withRS	1.2.840.1135
12	dsig-more#sha512-rsa-MGF1	AandMGF1	49.1.1.10

A.2. Key Management Algorithm Identifier Cross-Reference

This section contains a table cross-referencing the JWE "alg" (algorithm) values defined in this specification with the equivalent identifiers used by other standards and software packages.

JWE	XML ENC	JCA	OID
RSA1_5	http://www.w3.org/2001/04/xmlenc#rsa-1_5	RSA/ECB/PKCS1Padding	1.2.840.1135 49.1.1.1
RSA-OAEP	http://www.w3.org/2001/04/xmlenc#rsa-oaep-mgf1p	RSA/ECB/OAEPWithSHA-1AndMGF1Padding	1.2.840.1135 49.1.1.7
ECDH-ES	http://www.w3.org/2009/xmlenc11#ECDH-ES		1.3.132.1.12
A128KW	http://www.w3.org/2001/04/xmlenc#kw-aes128		2.16.840.1.1 01.3.4.1.5
A192KW	http://www.w3.org/2001/04/xmlenc#kw-aes192		2.16.840.1.1 01.3.4.1.25
A256KW	http://www.w3.org/2001/04/xmlenc#kw-aes256		2.16.840.1.1 01.3.4.1.45

A.3. Content Encryption Algorithm Identifier Cross-Reference

This section contains a table cross-referencing the JWE "enc" (encryption algorithm) values defined in this specification with the equivalent identifiers used by other standards and software packages.

For the composite algorithms "A128CBC-HS256", "A192CBC-HS384", and "A256CBC-HS512", the corresponding AES CBC algorithm identifiers are listed.

JWE	XML ENC	JCA	OID
A128CBC-HS256	http://www.w3.org/2001/04/xmlenc#aes128-cbc	AES/CBC/PKCS5Padding	2.16.840.1.101.3.4.1.2
A192CBC-HS384	http://www.w3.org/2001/04/xmlenc#aes192-cbc	AES/CBC/PKCS5Padding	2.16.840.1.101.3.4.1.22
A256CBC-HS512	http://www.w3.org/2001/04/xmlenc#aes256-cbc	AES/CBC/PKCS5Padding	2.16.840.1.101.3.4.1.42
A128GCM	http://www.w3.org/2009/xmlenc11#aes128-gcm	AES/GCM/NoPadding	2.16.840.1.101.3.4.1.6
A192GCM	http://www.w3.org/2009/xmlenc11#aes192-gcm	AES/GCM/NoPadding	2.16.840.1.101.3.4.1.26

| A256GCM | http://www.w3.org/2009/ xmlenc11#aes256-gcm | AES/GCM/NoPadding | 2.16.840.1.101.3.4.1.46 |

Appendix B. 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 5.2. 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 5.2. All values are hexadecimal.

B.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
          4b 88 51 ff b5 98 f7 f8 00 74 b9 47 3c 82 e2 db

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
```

B.2. Test Cases for AES_192_CBC_HMAC_SHA_384

```
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

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

ENC_KEY = 18 19 1a 1b 1c 1d 1e 1f 20 21 22 23 24 25 26 27
          28 29 2a 2b 2c 2d 2e 2f

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 =       ea 65 da 6b 59 e6 1e db 41 9b e6 2d 19 71 2a e5
          d3 03 ee b5 00 52 d0 df d6 69 7f 77 22 4c 8e db
          00 0d 27 9b dc 14 c1 07 26 54 bd 30 94 42 30 c6
          57 be d4 ca 0c 9f 4a 84 66 f2 2b 22 6d 17 46 21
          4b f8 cf c2 40 0a dd 9f 51 26 e4 79 66 3f c9 0b
          3b ed 78 7a 2f 0f fc bf 39 04 be 2a 64 1d 5c 21
          05 bf e5 91 ba e2 3b 1d 74 49 e5 32 ee f6 0a 9a
          c8 bb 6c 6b 01 d3 5d 49 78 7b cd 57 ef 48 49 27
          f2 80 ad c9 1a c0 c4 e7 9c 7b 11 ef c6 00 54 e3

M =       84 90 ac 0e 58 94 9b fe 51 87 5d 73 3f 93 ac 20
          75 16 80 39 cc c7 33 d7 45 94 f8 86 b3 fa af d4
          86 f2 5c 71 31 e3 28 1e 36 c7 a2 d1 30 af de 57

T =       84 90 ac 0e 58 94 9b fe 51 87 5d 73 3f 93 ac 20
          75 16 80 39 cc c7 33 d7
```

B.3. 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 C. Example ECDH-ES Key Agreement Computation

This example uses ECDH-ES Key Agreement and the Concat KDF to derive the Content Encryption Key (CEK) in the manner described in Section 4.6. In this example, the ECDH-ES Direct Key Agreement mode ("alg" value "ECDH-ES") is used to produce an agreed upon key for AES GCM with a 128 bit key ("enc" value "A128GCM").

In this example, a sender Alice is encrypting content to a recipient Bob. The sender (Alice) generates an ephemeral key for the key agreement computation. Alice's ephemeral key (in JWK format) used for the key agreement computation in this example (including the private part) is:

```
{"kty":"EC",
 "crv":"P-256",
 "x":"gI0GAILBdu7T53akrFmMyGcsF3n5dO7MmwNBHKW5SV0",
 "y":"SLW_xSffzlPWrHEVI30DHM_4egVwt3NQqeUD7nMFpps",
 "d":"0_NxaRPUMQoAJt50Gz8YiTr8gRTwyEaCumd-MToTmIo"
}
```

The recipient's (Bob's) key (in JWK format) used for the key agreement computation in this example (including the private part) is:

```
{"kty":"EC",
 "crv":"P-256",
 "x":"weNJy2HscCSM6AEDTDg04biOvhFhyyWvOHQfeF_PxMQ",
 "y":"e8lnCO-AlStT-NJVX-crhB7QRYhiix03illJOVAOyck",
 "d":"VEmDZpDXXK8p8N0Cndsxs924q6nS1RXFASRl6BfUqdw"
}
```

Header Parameter values used in this example are as follows. In this example, the "apu" (agreement PartyUInfo) parameter value is the base64url encoding of the UTF-8 string "Alice" and the "apv" (agreement PartyVInfo) parameter value is the base64url encoding of the UTF-8 string "Bob". The "epk" parameter is used to communicate the sender's (Alice's) ephemeral public key value to the recipient (Bob).

```
  {"alg":"ECDH-ES",
   "enc":"A128GCM",
   "apu":"QWxpY2U",
   "apv":"Qm9i",
   "epk":
    {"kty":"EC",
     "crv":"P-256",
     "x":"gI0GAILBdu7T53akrFmMyGcsF3n5dO7MmwNBHKW5SV0",
     "y":"SLW_xSffzlPWrHEVI30DHM_4egVwt3NQqeUD7nMFpps"
    }
  }
```

The resulting Concat KDF [NIST.800-56A] parameter values are:

Z This is set to the ECDH-ES key agreement output. (This value is often not directly exposed by libraries, due to NIST security requirements, and only serves as an input to a KDF.) In this example, Z is the octet sequence: [158, 86, 217, 29, 129, 113, 53, 211, 114, 131, 66, 131, 191, 132, 38, 156, 251, 49, 110, 163, 218, 128, 106, 72, 246, 218, 167, 121, 140, 254, 144, 196].

keydatalen This value is 128 - the number of bits in the desired output key (because "A128GCM" uses a 128 bit key).

AlgorithmID This is set to the octets representing the 32 bit big endian value 7 - [0, 0, 0, 7] - the number of octets in the AlgorithmID content "A128GCM", followed, by the octets representing the UTF-8 string "A128GCM" - [65, 49, 50, 56, 71, 67, 77].

PartyUInfo This is set to the octets representing the 32 bit big endian value 5 - [0, 0, 0, 5] - the number of octets in the PartyUInfo content "Alice", followed, by the octets representing the UTF-8 string "Alice" - [65, 108, 105, 99, 101].

PartyVInfo This is set to the octets representing the 32 bit big endian value 3 - [0, 0, 0, 3] - the number of octets in the PartyUInfo content "Bob", followed, by the octets representing the UTF-8 string "Bob" - [66, 111, 98].

SuppPubInfo This is set to the octets representing the 32 bit big endian value 128 - [0, 0, 0, 128] - the keydatalen value.

SuppPrivInfo This is set to the empty octet sequence.

Concatenating the parameters AlgorithmID through SuppPubInfo results in an OtherInfo value of:

[0, 0, 0, 7, 65, 49, 50, 56, 71, 67, 77, 0, 0, 0, 5, 65, 108, 105, 99, 101, 0, 0, 0, 3, 66, 111, 98, 0, 0, 0, 128]

Concatenating the round number 1 ([0, 0, 0, 1]), Z, and the OtherInfo value results in the Concat KDF round 1 hash input of:
[0, 0, 0, 1,
158, 86, 217, 29, 129, 113, 53, 211, 114, 131, 66, 131, 191, 132, 38, 156, 251, 49, 110, 163, 218, 128, 106, 72, 246, 218, 167, 121, 140, 254, 144, 196,
0, 0, 0, 7, 65, 49, 50, 56, 71, 67, 77, 0, 0, 0, 5, 65, 108, 105, 99, 101, 0, 0, 0, 3, 66, 111, 98, 0, 0, 0, 128]

The resulting derived key, which is the first 128 bits of the round 1 hash output is:
[86, 170, 141, 234, 248, 35, 109, 32, 92, 34, 40, 205, 113, 167, 16, 26]

The base64url encoded representation of this derived key is:

VqqN6vgjbSBcIijNcacQGg

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.

Matt Miller wrote Using JavaScript Object Notation (JSON) Web Encryption (JWE) for Protecting JSON Web Key (JWK) Objects [I-D.miller-jose-jwe-protected-jwk], which the password-based encryption content of this draft is based upon.

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, Vladimir Dzhuvinov, Yaron Y. Goland, Dick Hardt, Jeff Hodges, Edmund Jay, James Manger, Matt Miller, 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]]

-20

- Replaced references to RFC 4627 with draft-ietf-json-rfc4627bis, addressing issue #90.

-19

- Used tables to show the correspondence between algorithm identifiers and algorithm descriptions and parameters in the algorithm definition sections, addressing issue #183.

- Changed the "Implementation Requirements" registry field names to "JOSE Implementation Requirements" to make it clear that these implementation requirements apply only to JWS and JWE implementations.

-18

- Changes to address editorial and minor issues #129, #134, #135, #158, #161, #185, #186, and #187.

- Added and used Description registry fields.

-17

- Explicitly named all the logical components of a JWS and JWE and defined the processing rules and serializations in terms of those components, addressing issues #60, #61, and #62.

- Removed processing steps in algorithm definitions that duplicated processing steps in JWS or JWE, addressing issue #56.

- Replaced verbose repetitive phases such as "base64url encode the octets of the UTF-8 representation of X" with mathematical

notation such as "BASE64URL(UTF8(X))".

- Terms used in multiple documents are now defined in one place and incorporated by reference. Some lightly used or obvious terms were also removed. This addresses issue #58.

- Changes to address minor issue #53.

-16

- Added a DataLen prefix to the AlgorithmID value in the Concat KDF computation.

- Added OIDs for encryption algorithms, additional signature algorithm OIDs, and additional XML DSIG/ENC URIs in the algorithm cross-reference tables.

- Changes to address editorial and minor issues #28, #36, #39, #52, #53, #55, #127, #128, #136, #137, #141, #150, #151, #152, and #155.

-15

- Changed statements about rejecting JWSs to statements about validation failing, addressing issue #35.

- Stated that changes of implementation requirements are only permitted on a Specification Required basis, addressing issue #38.

- Made "oct" a required key type, addressing issue #40.

- Updated the example ECDH-ES key agreement values.

- Changes to address editorial and minor issues #34, #37, #49, #63, #123, #124, #125, #130, #132, #133, #138, #139, #140, #142, #143, #144, #145, #148, #149, #150, and #162.

-14

- Removed "PBKDF2" key type and added "p2s" and "p2c" header parameters for use with the PBES2 algorithms.

- Made the RSA private key parameters that are there to enable optimizations be RECOMMENDED rather than REQUIRED.

- Added algorithm identifiers for AES algorithms using 192 bit keys and for RSASSA-PSS using HMAC SHA-384.

- Added security considerations about key lifetimes, addressing issue #18.

- Added an example ECDH-ES key agreement computation.

-13

- Added key encryption with AES GCM as specified in draft-jones-jose-aes-gcm-key-wrap-01, addressing issue #13.

- Added security considerations text limiting the number of times that an AES GCM key can be used for key encryption or direct encryption, per Section 8.3 of NIST SP 800-38D, addressing issue #28.

- Added password-based key encryption as specified in draft-miller-jose-jwe-protected-jwk-02.

-12

- 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.

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

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

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

-11

- 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.

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

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

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

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

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

-10

- 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

- 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.

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

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

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

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

- 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.

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

-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 Authentication Tag.
- 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".

Author's Address

Michael B. Jones
Microsoft

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