

JOSE Working Group
Internet-Draft
Intended status: Standards Track
Expires: April 10, 2014

M. Jones
Microsoft
October 7, 2013

JSON Web Algorithms (JWA)
draft-ietf-jose-json-web-algorithms-17

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 April 10, 2014.

Copyright Notice

Copyright (c) 2013 IETF Trust and the persons identified as the document authors. All rights reserved.

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

Table of Contents

1. Introduction	5
1.1. Notational Conventions	5
2. Terminology	5
3. Cryptographic Algorithms for Digital Signatures and MACs	6
3.1. "alg" (Algorithm) Header Parameter Values for JWS	6
3.2. HMAC with SHA-2 Functions	7
3.3. Digital Signature with RSASSA-PKCS1-V1_5	8
3.4. Digital Signature with ECDSA	9
3.5. Digital Signature with RSASSA-PSS	10
3.6. Using the Algorithm "none"	11
4. Cryptographic Algorithms for Encryption	11
4.1. "alg" (Algorithm) Header Parameter Values for JWE	11
4.2. "enc" (Encryption Method) Header Parameter Values for JWE	16
4.3. Key Encryption with RSAES-PKCS1-V1_5	17
4.4. Key Encryption with RSAES OAEP	17
4.5. Key Wrapping with AES Key Wrap	18
4.6. Direct Encryption with a Shared Symmetric Key	18
4.7. Key Agreement with Elliptic Curve Diffie-Hellman Ephemeral Static (ECDH-ES)	18
4.7.1. Header Parameters Used for ECDH Key Agreement	19
4.7.1.1. "epk" (Ephemeral Public Key) Header Parameter	19
4.7.1.2. "apu" (Agreement PartyUInfo) Header Parameter	19
4.7.1.3. "apv" (Agreement PartyVInfo) Header Parameter	19
4.7.2. Key Derivation for ECDH Key Agreement	19
4.8. Key Encryption with AES GCM	21
4.8.1. Header Parameters Used for AES GCM Key Encryption	21
4.8.1.1. "iv" (Initialization Vector) Header Parameter	21
4.8.1.2. "tag" (Authentication Tag) Header Parameter	22
4.9. Key Encryption with PBES2	22
4.9.1. Header Parameters Used for PBES2 Key Encryption	22
4.9.1.1. "p2s" (PBES2 salt) Parameter	22
4.9.1.2. "p2c" (PBES2 count) Parameter	23
4.10. AES_CBC_HMAC_SHA2 Algorithms	23
4.10.1. Conventions Used in Defining AES_CBC_HMAC_SHA2	23
4.10.2. Generic AES_CBC_HMAC_SHA2 Algorithm	23
4.10.2.1. AES_CBC_HMAC_SHA2 Encryption	24
4.10.2.2. AES_CBC_HMAC_SHA2 Decryption	25
4.10.3. AES_128_CBC_HMAC_SHA_256	26
4.10.4. AES_192_CBC_HMAC_SHA_384	26
4.10.5. AES_256_CBC_HMAC_SHA_512	27
4.10.6. Plaintext Encryption with AES_CBC_HMAC_SHA2	27
4.11. Plaintext Encryption with AES GCM	27
5. Cryptographic Algorithms for Keys	28
5.1. "kty" (Key Type) Parameter Values	28
5.2. Parameters for Elliptic Curve Keys	28

5.2.1. Parameters for Elliptic Curve Public Keys	28
5.2.1.1. "crv" (Curve) Parameter	29
5.2.1.2. "x" (X Coordinate) Parameter	29
5.2.1.3. "y" (Y Coordinate) Parameter	29
5.2.2. Parameters for Elliptic Curve Private Keys	29
5.2.2.1. "d" (ECC Private Key) Parameter	29
5.3. Parameters for RSA Keys	30
5.3.1. Parameters for RSA Public Keys	30
5.3.1.1. "n" (Modulus) Parameter	30
5.3.1.2. "e" (Exponent) Parameter	30
5.3.2. Parameters for RSA Private Keys	30
5.3.2.1. "d" (Private Exponent) Parameter	30
5.3.2.2. "p" (First Prime Factor) Parameter	31
5.3.2.3. "q" (Second Prime Factor) Parameter	31
5.3.2.4. "dp" (First Factor CRT Exponent) Parameter	31
5.3.2.5. "dq" (Second Factor CRT Exponent) Parameter	31
5.3.2.6. "qi" (First CRT Coefficient) Parameter	31
5.3.2.7. "oth" (Other Primes Info) Parameter	32
5.4. Parameters for Symmetric Keys	32
5.4.1. "k" (Key Value) Parameter	33
6. IANA Considerations	33
6.1. JSON Web Signature and Encryption Algorithms Registry	34
6.1.1. Template	34
6.1.2. Initial Registry Contents	35
6.2. JWE Header Parameter Names Registration	39
6.2.1. Registry Contents	39
6.3. JSON Web Encryption Compression Algorithms Registry	40
6.3.1. Registration Template	40
6.3.2. Initial Registry Contents	41
6.4. JSON Web Key Types Registry	41
6.4.1. Registration Template	41
6.4.2. Initial Registry Contents	42
6.5. JSON Web Key Parameters Registration	43
6.5.1. Registry Contents	43
6.6. JSON Web Key Elliptic Curve Registry	45
6.6.1. Registration Template	45
6.6.2. Initial Registry Contents	46
7. Security Considerations	46
7.1. Algorithms and Key Sizes will be Deprecated	46
7.2. Key Lifetimes	47
7.3. RSAES-PKCS1-v1_5 Security Considerations	47
7.4. AES GCM Security Considerations	47
7.5. Plaintext JWS Security Considerations	47
7.6. Denial of Service Attacks	48
7.7. Reusing Key Material when Encrypting Keys	48
7.8. Password Considerations	48
8. Internationalization Considerations	49
9. References	49

9.1. Normative References	49
9.2. Informative References	51
Appendix A. Digital Signature/MAC Algorithm Identifier Cross-Reference	53
Appendix B. Encryption Algorithm Identifier Cross-Reference . . .	53
Appendix C. Test Cases for AES_CBC_HMAC_SHA2 Algorithms	54
C.1. Test Cases for AES_128_CBC_HMAC_SHA_256	55
C.2. Test Cases for AES_192_CBC_HMAC_SHA_384	56
C.3. Test Cases for AES_256_CBC_HMAC_SHA_512	57
Appendix D. Example ECDH-ES Key Agreement Computation	58
Appendix E. Acknowledgements	60
Appendix F. Document History	61
Author's Address	68

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) [RFC4627] 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)", "JSON Text Object", "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 hash algorithm	Required
HS384	HMAC using SHA-384 hash algorithm	Optional
HS512	HMAC using SHA-512 hash algorithm	Optional
RS256	RSASSA-PKCS-v1_5 using SHA-256 hash algorithm	Recommended
RS384	RSASSA-PKCS-v1_5 using SHA-384 hash algorithm	Optional
RS512	RSASSA-PKCS-v1_5 using SHA-512 hash algorithm	Optional
ES256	ECDSA using P-256 curve and SHA-256 hash algorithm	Recommended+
ES384	ECDSA using P-384 curve and SHA-384 hash algorithm	Optional
ES512	ECDSA using P-521 curve and SHA-512 hash algorithm	Optional
PS256	RSASSA-PSS using SHA-256 hash algorithm and MGF1 mask generation function with SHA-256	Optional
PS384	RSASSA-PSS using SHA-384 hash algorithm and MGF1 mask generation function with SHA-384	Optional
PS512	RSASSA-PSS using SHA-512 hash algorithm and MGF1 mask generation function with SHA-512	Optional
none	No digital signature or MAC value included	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 for a table cross-referencing the digital signature and MAC "alg" (algorithm) values used in this specification with the equivalent identifiers used by other standards and software packages.

3.2. 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]. This section defines the use of the HMAC SHA-256, HMAC SHA-384, and HMAC SHA-512 functions [SHS]. The "alg" (algorithm) Header Parameter values "HS256", "HS384", and "HS512" are used in the JWS Header to indicate that the JWS Signature contains an HMAC value using the respective hash function.

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

The HMAC SHA-256 MAC is generated per RFC 2104, using SHA-256 as the hash algorithm "H", using the JWS Signing Input as the "text" value, and using the shared key. The HMAC output value is the JWS Signature.

The HMAC SHA-256 MAC for a JWS is validated by computing an HMAC value per RFC 2104, using SHA-256 as the hash algorithm "H", using the 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 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-256, SHA-384, or SHA-512 [SHS] as the hash functions. The "alg" (algorithm) header parameter values "RS256", "RS384", and "RS512" are used in the JWS Header to indicate that the JWS Signature contains a RSASSA-PKCS1-V1_5 digital signature using the respective hash function.

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

The RSASSA-PKCS1-V1_5 SHA-256 digital signature is generated as follows: 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 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 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 "alg" (algorithm) Header Parameter values "ES256", "ES384", and "ES512" are used in the JWS Header to indicate that the JWS Signature contains a base64url encoded ECDSA P-256 SHA-256, ECDSA P-384 SHA-384, or ECDSA P-521 SHA-512 digital signature, respectively.

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

1. Generate a digital signature of the 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 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 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 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, always using the same hash function for both the RSASSA-PSS hash function and the MGF1 hash function. Use of SHA-256, SHA-384, and SHA-512 as these hash functions is defined. 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. The "alg" (algorithm) header parameter values "PS256", "PS384", and "PS512" are used in the JWS Header to indicate that the JWS Signature contains a base64url encoded RSASSA-PSS digital signature using the respective hash function in both roles.

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

The RSASSA-PSS SHA-256 digital signature is generated as follows: 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 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 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". Plaintext JWSs MUST use the "alg" value "none", and are formatted identically to other JWSs, but with the empty string for its JWS Signature value. See Section 7.5 for security considerations associated with using this algorithm.

4. Cryptographic Algorithms for Encryption

JWE uses cryptographic algorithms to encrypt the Content Encryption Key (CEK) and the Plaintext.

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

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

alg Parameter Value	Key Management Algorithm	Additional Header Parameters	Implementation Requirements
RSA1_5	RSAES-PKCS1-V1_5[RFC3447]	(none)	Required
RSA-OAEP	RSAES using Optimal Asymmetric Encryption Padding (OAEP) [RFC3447], with the default parameters specified by RFC 3447 in Section A.2.1	(none)	Optional
A128KW	Advanced Encryption Standard (AES) Key Wrap Algorithm [RFC3394] using the default initial value specified in Section 2.2.3.1 and using 128 bit keys	(none)	Recommended
A192KW	AES Key Wrap Algorithm using the default initial value specified in Section 2.2.3.1 and using 192 bit keys	(none)	Optional
A256KW	AES Key Wrap Algorithm using the default initial value specified in Section 2.2.3.1 and using 256 bit keys	(none)	Recommended

dir	Direct use of a shared symmetric key as the Content Encryption Key (CEK) for the content encryption step (rather than using the symmetric key to wrap the CEK)	(none)	Recommended
ECDH-ES	Elliptic Curve Diffie-Hellman Ephemeral Static [RFC6090] key agreement using the Concat KDF, as defined in Section 5.8.1 of [NIST.800-56A], with the agreed-upon key being used directly as the Content Encryption Key (CEK) (rather than being used to wrap the CEK), as specified in Section 4.7	"epk", "apu", "apv"	Recommended+

ECDH-ES+A128KW	Elliptic Curve Diffie-Hellman Ephemeral Static key agreement per "ECDH-ES" and Section 4.7, where the agreed-upon key is used to wrap the Content Encryption Key (CEK) with the "A128KW" function (rather than being used directly as the CEK)	"epk", "apu", "apv"	Recommended
ECDH-ES+A192KW	Elliptic Curve Diffie-Hellman Ephemeral Static key agreement, where the agreed-upon key is used to wrap the Content Encryption Key (CEK) with the "A192KW" function (rather than being used directly as the CEK)	"epk", "apu", "apv"	Optional

ECDH-ES+A256KW	Elliptic Curve Diffie-Hellman Ephemeral Static key agreement, where the agreed-upon key is used to wrap the Content Encryption Key (CEK) with the "A256KW" function (rather than being used directly as the CEK)	"epk", "apu", "apv"	Recommended
A128GCMKW	AES in Galois/Counter Mode (GCM) [AES] [NIST.800-38D] using 128 bit keys	"iv", "tag"	Optional
A192GCMKW	AES GCM using 192 bit keys	"iv", "tag"	Optional
A256GCMKW	AES GCM using 256 bit keys	"iv", "tag"	Optional
PBES2-HS256+A128KW	PBES2 [RFC2898] with HMAC SHA-256 as the PRF and AES Key Wrap [RFC3394] using 128 bit keys for the encryption scheme	"p2s", "p2c"	Optional
PBES2-HS384+A192KW	PBES2 with HMAC SHA-256 as the PRF and AES Key Wrap using 192 bit keys for the encryption scheme	"p2s", "p2c"	Optional

PBES2-HS512+A256KW	PBES2 with HMAC SHA-256 as the PRF and AES Key Wrap using 256 bit keys for the encryption scheme	"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.

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

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

enc Parameter Value	Content Encryption Algorithm	Additional Header Parameters	Implementatio nRequirements
A128CBC-HS256	The AES_128_CBC_HMAC_SHA_256 authenticated encryption algorithm, as defined in Section 4.10.3. This algorithm uses a 256 bit key.	(none)	Required
A192CBC-HS384	The AES_192_CBC_HMAC_SHA_384 authenticated encryption algorithm, as defined in Section 4.10.4. This algorithm uses a 384 bit key.	(none)	Optional

A256CBC-HS512	The AES_256_CBC_HMAC_SHA_512 authenticated encryption algorithm, as defined in Section 4.10.5. This algorithm uses a 512 bit key.	(none)	Required
A128GCM	AES in Galois/Counter Mode (GCM) [AES] [NIST.800-38D] using 128 bit keys	(none)	Recommended
A192GCM	AES GCM using 192 bit keys	(none)	Optional
A256GCM	AES GCM using 256 bit keys	(none)	Recommended

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

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

4.3. Key Encryption with RSAES-PKCS1-V1_5

This section defines the specifics of encrypting a JWE CEK with RSAES-PKCS1-V1_5 [RFC3447]. The "alg" Header Parameter value "RSA1_5" is used in this case.

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

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

4.4. Key Encryption with RSAES OAEP

This section defines the specifics of encrypting a JWE CEK with RSAES using Optimal Asymmetric Encryption Padding (OAEP) [RFC3447], with the default parameters specified by RFC 3447 in Section A.2.1. (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 in this case.

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

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

4.5. Key Wrapping with AES Key Wrap

This section defines the specifics of encrypting a JWE CEK with the Advanced Encryption Standard (AES) Key Wrap Algorithm [RFC3394] using the default initial value specified in Section 2.2.3.1 using 128, 192, or 256 bit keys. The "alg" Header Parameter values "A128KW", "A192KW", or "A256KW" are respectively used in this case.

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

4.6. Direct Encryption with a Shared Symmetric Key

This section defines the specifics of directly performing symmetric key encryption without performing a key wrapping step. In this case, the shared symmetric key is used directly as the Content Encryption Key (CEK) value for the "enc" algorithm. An empty octet sequence is used as the JWE Encrypted Key value. The "alg" Header Parameter value "dir" is used in this case.

Refer to the security considerations on key lifetimes in Section 7.2 and AES GCM in Section 7.4 when considering utilizing direct encryption.

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

This section defines the specifics of key agreement with Elliptic Curve Diffie-Hellman Ephemeral Static [RFC6090], 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.

The "alg" Header Parameter value "ECDH-ES" is used in the Direct Key Agreement mode and the values "ECDH-ES+A128KW", "ECDH-ES+A192KW", or "ECDH-ES+A256KW" are used in the Key Agreement with Key Wrapping mode.

In the Direct Key Agreement case, the output of the Concat KDF MUST be a key of the same length as that used by the "enc" algorithm; in this case, the empty octet sequence is used as the JWE Encrypted Key

value. In the Key Agreement with Key Wrapping case, the output of the Concat KDF MUST be a key of the length needed for the specified key wrapping algorithm, one of 128, 192, or 256 bits respectively.

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

4.7.1. Header Parameters Used for ECDH Key Agreement

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

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

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

4.7.1.2. "apu" (Agreement PartyUInfo) Header Parameter

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

4.7.1.3. "apv" (Agreement PartyVInfo) Header Parameter

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

4.7.2. Key Derivation for ECDH Key Agreement

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

Key derivation is performed using the Concat KDF, as defined in

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

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

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

PartyUIInfo The PartyUIInfo 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 PartyUIInfo) 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 MUST specify what values should be used in the "apu" and "apv" parameters. The "apu" and "apv" values MUST be distinct. 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 D for an example key agreement computation using this method.

4.8. 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] using 128, 192, or 256 bit keys. The "alg" Header Parameter values "A128GCMKW", "A192GCMKW", or "A256GCMKW" are respectively used in this case.

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.

4.8.1. Header Parameters Used for AES GCM Key Encryption

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

4.8.1.1. "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 is REQUIRED and MUST be understood and processed by implementations when these algorithms are used.

4.8.1.2. "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 is REQUIRED and MUST be understood and processed by implementations when these algorithms are used.

4.9. Key Encryption with PBES2

The "PBES2-HS256+A128KW", "PBES2-HS384+A192KW", and "PBES2-HS512+A256KW" composite algorithms are used to perform password-based encryption of a JWE CEK, by first deriving a key encryption key from a user-supplied password, then encrypting the JWE CEK using the derived key. These algorithms are PBES2 schemes as specified in Section 6.2 of [RFC2898].

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 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.

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

4.9.1. Header Parameters Used for PBES2 Key Encryption

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

4.9.1.1. "p2s" (PBES2 salt) Parameter

The "p2s" (PBES2 salt) Header Parameter contains the PBKDF2 salt value, encoded using base64url. This Header Parameter is REQUIRED 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.

4.9.1.2. "p2c" (PBES2 count) Parameter

The "p2c" (PBES2 count) Header Parameter contains the PBKDF2 iteration count, represented as a positive integer. This Header Parameter is REQUIRED 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.

4.10. 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 C.

These algorithms are based upon Authenticated Encryption with AES-CBC and HMAC-SHA [I-D.mcgregw-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.mcgregw-aead-aes-cbc-hmac-sha2], and can be used to implement those algorithms.

4.10.1. 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.

4.10.2. Generic AES_CBC_HMAC_SHA2 Algorithm

This section defines AES_CBC_HMAC_SHA2 in a manner that is independent of the AES CBC key size or hash function to be used. Section 4.10.2.1 and Section 4.10.2.2 define the generic encryption

and decryption algorithms. Section 4.10.3 and Section 4.10.5 define instances of AES_CBC_HMAC_SHA2 that specify those details.

4.10.2.1. AES_CBC_HMAC_SHA2 Encryption

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

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

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

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

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

Here we denote the number of octets in the MAC_KEY as MAC_KEY_LEN, and the number of octets in ENC_KEY as ENC_KEY_LEN; the values of these parameters are specified by the AEAD algorithms (in Section 4.10.3 and Section 4.10.5). 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.
```

4.10.2.2. AES_CBC_HMAC_SHA2 Decryption

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

1. The secondary keys MAC_KEY and ENC_KEY are generated from the input key K as in Step 1 of Section 4.10.2.1.
2. The integrity and authenticity of A and E are checked by computing an HMAC with the inputs as in Step 5 of Section 4.10.2.1. The value T, from the previous step, is compared to the first MAC_KEY length bits of the HMAC output. If those values are identical, then A and E are considered valid, and processing is continued. Otherwise, all of the data used in the MAC validation are discarded, and the AEAD decryption operation returns an indication that it failed, and the operation

halts. (But see Section 10 of [JWE] for security considerations on thwarting timing attacks.)

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

4.10.3. AES_128_CBC_HMAC_SHA_256

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

The input key K is 32 octets long.

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

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

4.10.4. 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:

A 192 bit AES CBC key is used instead of 128.

SHA-384 is used in HMAC instead of SHA-256.

ENC_KEY_LEN is 24 octets instead of 16.

MAC_KEY_LEN is 24 octets instead of 16.

The length of the input key K is 48 octets instead of 32.

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

4.10.5. 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:

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

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

ENC_KEY_LEN is 32 octets instead of 16.

MAC_KEY_LEN is 32 octets instead of 16.

The length of the input key K is 64 octets instead of 32.

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

4.10.6. Plaintext Encryption with AES_CBC_HMAC_SHA2

The algorithm value "A128CBC-HS256" is used as the "alg" value when using AES_128_CBC_HMAC_SHA_256 with JWE. The algorithm value "A192CBC-HS384" is used as the "alg" value when using AES_192_CBC_HMAC_SHA_384 with JWE. The algorithm value "A256CBC-HS512" is used as the "alg" value when using AES_256_CBC_HMAC_SHA_512 with JWE.

4.11. Plaintext Encryption with AES GCM

This section defines the specifics of encrypting the JWE Plaintext with Advanced Encryption Standard (AES) in Galois/Counter Mode (GCM) [AES] [NIST.800-38D] using 128, 192, or 256 bit keys. 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.

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

5. 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.

5.1. "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.

5.2. Parameters for Elliptic Curve Keys

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

5.2.1. 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:

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

SEC1 point compression is not supported for any values.

5.2.1.1. "crv" (Curve) Parameter

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

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

These values are registered in the IANA JSON Web Key Elliptic Curve registry defined in Section 6.6. 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.

5.2.1.2. "x" (X Coordinate) Parameter

The "x" (x coordinate) member contains the x coordinate for the elliptic curve point. It is represented as the base64url encoding of the 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.

5.2.1.3. "y" (Y Coordinate) Parameter

The "y" (y coordinate) member contains the y coordinate for the elliptic curve point. It is represented as the base64url encoding of the 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.

5.2.2. 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.

5.2.2.1. "d" (ECC Private Key) Parameter

The "d" (ECC private key) member contains the Elliptic Curve private key value. It is represented as the base64url encoding of the 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 $\text{ceiling}(\log\text{-base-2}(n)/8)$ octets (where n is the order of the curve).

5.3. Parameters for RSA Keys

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

5.3.1. Parameters for RSA Public Keys

The following members MUST be present for RSA public keys.

5.3.1.1. "n" (Modulus) Parameter

The "n" (modulus) member contains the modulus value for the RSA public key. It is represented as the base64url encoding of the value's unsigned big endian representation as an octet sequence. The octet sequence MUST utilize the minimum number of octets to represent the value.

5.3.1.2. "e" (Exponent) Parameter

The "e" (exponent) member contains the exponent value for the RSA public key. It is represented as the base64url encoding of the value's unsigned big endian representation as an octet sequence. The 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].

5.3.2. 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.

5.3.2.1. "d" (Private Exponent) Parameter

The "d" (private exponent) member contains the private exponent value for the RSA private key. It is represented as the base64url encoding

of the value's unsigned big endian representation as an octet sequence. The octet sequence MUST utilize the minimum number of octets to represent the value.

5.3.2.2. "p" (First Prime Factor) Parameter

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

5.3.2.3. "q" (Second Prime Factor) Parameter

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

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

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

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

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

5.3.2.6. "qi" (First CRT Coefficient) Parameter

The "dp" (first CRT coefficient) member contains the Chinese Remainder Theorem (CRT) coefficient of the second factor, a positive integer. It is represented as the base64url encoding of the value's unsigned big endian representation as an octet sequence. The octet sequence MUST utilize the minimum number of octets to represent the value.

5.3.2.7. "oth" (Other Primes Info) Parameter

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

5.3.2.7.1. "r" (Prime Factor)

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

5.3.2.7.2. "d" (Factor CRT Exponent)

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

5.3.2.7.3. "t" (Factor CRT Coefficient)

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

5.4. 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.

5.4.1. "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.

6. IANA Considerations

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

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

Registration requests must be sent to the [TBD]@ietf.org mailing list for review and comment, with an appropriate subject (e.g., "Request for access token type: example"). [[Note to 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).

6.1. JSON Web Signature and Encryption Algorithms Registry

This specification establishes the IANA JSON Web Signature and Encryption Algorithms registry for values of the JWS and JWE "alg" (algorithm) and "enc" (encryption method) Header Parameters. The registry records the algorithm name, the algorithm usage locations from the set "alg" and "enc", implementation requirements, and a reference to the specification that defines it. The same algorithm name MAY be registered multiple times, provided that the sets of usage locations are disjoint.

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.

6.1.1. 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 Usage Location(s):

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

Implementation Requirements:

The algorithm implementation requirements, which must be one the words Required, Recommended, Optional, or Deprecated. Optionally, the word can be followed by a "+" or "-". The use of "+" indicates that the requirement strength is likely to be increased in a future version of the specification. The use of "-" indicates that the requirement strength is likely to be decreased in a future version of the specification.

Change Controller:

For Standards Track RFCs, state "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.

6.1.2. Initial Registry Contents

- o Algorithm Name: "HS256"
- o Algorithm Usage Location(s): "alg"
- o Implementation Requirements: Required
- o Change Controller: IESG
- o Specification Document(s): Section 3.1 of [[this document]]

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

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

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

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

- o Algorithm Name: "RS512"
- o Algorithm Usage Location(s): "alg"
- o Implementation Requirements: Optional

- o Change Controller: IESG
- o Specification Document(s): Section 3.1 of [[this document]]

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

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

- o Algorithm Name: "ES512"
- o Algorithm Usage Location(s): "alg"
- o Implementation Requirements: Optional
- o Change Controller: IESG
- o Specification Document(s): Section 3.1 of [[this document]]

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

- o Algorithm Name: "PS384"
- o Algorithm Usage Location(s): "alg"
- o Implementation Requirements: Optional
- o Change Controller: IESG
- o Specification Document(s): Section 3.1 of [[this document]]

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

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

- o Algorithm Name: "RSA1_5"

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

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

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

- o Algorithm Name: "A192KW"
- o Algorithm Usage Location(s): "alg"
- o Implementation Requirements: Optional
- o Change Controller: IESG
- o Specification Document(s): Section 4.1 of [[this document]]

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

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

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

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

- o Algorithm Name: "ECDH-ES+A192KW"
- o Algorithm Usage Location(s): "alg"
- o Implementation Requirements: Optional
- o Change Controller: IESG
- o Specification Document(s): Section 4.1 of [[this document]]

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

- o Algorithm Name: "A128GCMKW"
- o Algorithm Usage Location(s): "alg"
- o Implementation Requirements: Optional
- o Change Controller: IESG
- o Specification Document(s): Section 4.8 of [[this document]]

- o Algorithm Name: "A192GCMKW"
- o Algorithm Usage Location(s): "alg"
- o Implementation Requirements: Optional
- o Change Controller: IESG
- o Specification Document(s): Section 4.8 of [[this document]]

- o Algorithm Name: "A256GCMKW"
- o Algorithm Usage Location(s): "alg"
- o Implementation Requirements: Optional
- o Change Controller: IESG
- o Specification Document(s): Section 4.8 of [[this document]]

- o Algorithm Name: "PBES2-HS256+A128KW"
- o Algorithm Usage Location(s): "alg"
- o Implementation Requirements: Optional
- o Change Controller: IESG
- o Specification Document(s): Section 4.9 of [[this document]]

- o Algorithm Name: "PBES2-HS384+A192KW"
- o Algorithm Usage Location(s): "alg"
- o Implementation Requirements: Optional
- o Change Controller: IESG
- o Specification Document(s): Section 4.9 of [[this document]]

- o Algorithm Name: "PBES2-HS512+A256KW"
- o Algorithm Usage Location(s): "alg"
- o Implementation Requirements: Optional
- o Change Controller: IESG

- o Specification Document(s): Section 4.9 of [[this document]]
- o Algorithm Name: "A128CBC-HS256"
- o Algorithm Usage Location(s): "enc"
- o Implementation Requirements: Required
- o Change Controller: IESG
- o Specification Document(s): Section 4.2 of [[this document]]
- o Algorithm Name: "A192CBC-HS384"
- o Algorithm Usage Location(s): "enc"
- o Implementation Requirements: Optional
- o Change Controller: IESG
- o Specification Document(s): Section 4.2 of [[this document]]
- o Algorithm Name: "A256CBC-HS512"
- o Algorithm Usage Location(s): "enc"
- o Implementation Requirements: Required
- o Change Controller: IESG
- o Specification Document(s): Section 4.2 of [[this document]]
- o Algorithm Name: "A128GCM"
- o Algorithm Usage Location(s): "enc"
- o Implementation Requirements: Recommended
- o Change Controller: IESG
- o Specification Document(s): Section 4.2 of [[this document]]
- o Algorithm Name: "A192GCM"
- o Algorithm Usage Location(s): "enc"
- o Implementation Requirements: Optional
- o Change Controller: IESG
- o Specification Document(s): Section 4.2 of [[this document]]
- o Algorithm Name: "A256GCM"
- o Algorithm Usage Location(s): "enc"
- o Implementation Requirements: Recommended
- o Change Controller: IESG
- o Specification Document(s): Section 4.2 of [[this document]]

6.2. JWE Header Parameter Names Registration

This specification registers the Header Parameter names defined in Section 4.7.1, Section 4.8.1, and Section 4.9.1 in the IANA JSON Web Signature and Encryption Header Parameters registry defined in [JWS].

6.2.1. Registry Contents

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

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

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

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

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

- o Header Parameter Name: "p2s"
- o Header Parameter Usage Location(s): JWE
- o Change Controller: IESG
- o Specification Document(s): Section 4.9.1.1 of [[this document]]

- o Header Parameter Name: "p2c"
- o Header Parameter Usage Location(s): JWE
- o Change Controller: IESG
- o Specification Document(s): Section 4.9.1.2 of [[this document]]

6.3. 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.

6.3.1. 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.

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.

6.3.2. Initial Registry Contents

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

6.4. 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.

6.4.1. 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.

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.

Implementation Requirements:

The key type implementation requirements, which must be one the words Required, Recommended, Optional, or Deprecated. Optionally, the word can be followed by a "+" or "-". The use of "+" indicates that the requirement strength is likely to be increased in a future version of the specification. The use of "-" indicates that the requirement strength is likely to be decreased in a future version of the specification.

Specification Document(s):

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

6.4.2. Initial Registry Contents

This specification registers the values defined in Section 5.1.

- o "kty" Parameter Value: "EC"
- o Implementation Requirements: Recommended+
- o Change Controller: IESG
- o Specification Document(s): Section 5.2 of [[this document]]

- o "kty" Parameter Value: "RSA"
- o Implementation Requirements: Required
- o Change Controller: IESG
- o Specification Document(s): Section 5.3 of [[this document]]

- o "kty" Parameter Value: "oct"
- o Implementation Requirements: Required

- o Change Controller: IESG
- o Specification Document(s): Section 5.4 of [[this document]]

6.5. JSON Web Key Parameters Registration

This specification registers the parameter names defined in Sections 5.2, 5.3, and 5.4 in the IANA JSON Web Key Parameters registry defined in [JWK].

6.5.1. Registry Contents

- o Parameter Name: "crv"
- o Used with "kty" Value(s): "EC"
- o Parameter Information Class: Public
- o Change Controller: IESG
- o Specification Document(s): Section 5.2.1.1 of [[this document]]

- o Parameter Name: "x"
- o Used with "kty" Value(s): "EC"
- o Parameter Information Class: Public
- o Change Controller: IESG
- o Specification Document(s): Section 5.2.1.2 of [[this document]]

- o Parameter Name: "y"
- o Used with "kty" Value(s): "EC"
- o Parameter Information Class: Public
- o Change Controller: IESG
- o Specification Document(s): Section 5.2.1.3 of [[this document]]

- o Parameter Name: "d"
- o Used with "kty" Value(s): "EC"
- o Parameter Information Class: Private
- o Change Controller: IESG
- o Specification Document(s): Section 5.2.2.1 of [[this document]]

- o Parameter Name: "n"
- o Used with "kty" Value(s): "RSA"
- o Parameter Information Class: Public
- o Change Controller: IESG
- o Specification Document(s): Section 5.3.1.1 of [[this document]]

- o Parameter Name: "e"
- o Used with "kty" Value(s): "RSA"
- o Parameter Information Class: Public
- o Change Controller: IESG
- o Specification Document(s): Section 5.3.1.2 of [[this document]]

- o Parameter Name: "d"
- o Used with "kty" Value(s): "RSA"
- o Parameter Information Class: Private
- o Change Controller: IESG
- o Specification Document(s): Section 5.3.2.1 of [[this document]]

- o Parameter Name: "p"
- o Used with "kty" Value(s): "RSA"
- o Parameter Information Class: Private
- o Change Controller: IESG
- o Specification Document(s): Section 5.3.2.2 of [[this document]]

- o Parameter Name: "q"
- o Used with "kty" Value(s): "RSA"
- o Parameter Information Class: Private
- o Change Controller: IESG
- o Specification Document(s): Section 5.3.2.3 of [[this document]]

- o Parameter Name: "dp"
- o Used with "kty" Value(s): "RSA"
- o Parameter Information Class: Private
- o Change Controller: IESG
- o Specification Document(s): Section 5.3.2.4 of [[this document]]

- o Parameter Name: "dq"
- o Used with "kty" Value(s): "RSA"
- o Parameter Information Class: Private
- o Change Controller: IESG
- o Specification Document(s): Section 5.3.2.5 of [[this document]]

- o Parameter Name: "qi"
- o Used with "kty" Value(s): "RSA"
- o Parameter Information Class: Private
- o Change Controller: IESG
- o Specification Document(s): Section 5.3.2.6 of [[this document]]

- o Parameter Name: "oth"
- o Used with "kty" Value(s): "RSA"
- o Parameter Information Class: Private
- o Change Controller: IESG
- o Specification Document(s): Section 5.3.2.7 of [[this document]]

- o Parameter Name: "k"
- o Used with "kty" Value(s): "oct"
- o Parameter Information Class: Private
- o Change Controller: IESG

- o Specification Document(s): Section 5.4.1 of [[this document]]

6.6. 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 5.2.1.1.

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.

6.6.1. 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.

Implementation Requirements:

The curve implementation requirements, which must be one the words Required, Recommended, Optional, or Deprecated. Optionally, the word can be followed by a "+" or "-". The use of "+" indicates that the requirement strength is likely to be increased in a future version of the specification. The use of "-" indicates that the requirement strength is likely to be decreased in a future version of the specification.

Change Controller:

For Standards Track RFCs, state "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.

6.6.2. Initial Registry Contents

- o Curve Name: "P-256"
- o Implementation Requirements: Recommended+
- o Change Controller: IESG
- o Specification Document(s): Section 5.2.1.1 of [[this document]]

- o Curve Name: "P-384"
- o Implementation Requirements: Optional
- o Change Controller: IESG
- o Specification Document(s): Section 5.2.1.1 of [[this document]]

- o Curve Name: "P-521"
- o Implementation Requirements: Optional
- o Change Controller: IESG
- o Specification Document(s): Section 5.2.1.1 of [[this document]]

7. Security Considerations

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

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

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.

7.1. 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 removed. Therefore, implementers and deployments must be prepared for this eventuality.

7.2. 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.

7.3. 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.

7.4. 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.

7.5. 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.

7.6. 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.

7.7. 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].

7.8. Password Considerations

While convenient for end users, 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-entry 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 (or larger) than the derived key length but less than the PRF's block size. Passwords larger than the PRF's block size are first hashed, which reduces an attacker's effective search space to the length of the hash algorithm (32 octets for HMAC SHA-256). It is RECOMMENDED that the password be no longer than 64 octets long for "PBES2-HS512+A256KW".

Still, care needs to be taken in where and how password-based encryption is used. Such algorithms MUST NOT be used where the attacker can make an indefinite number of attempts to circumvent the protection.

8. 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.

9. References

9.1. 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.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), October 2013.
- [JWK] Jones, M., "JSON Web Key (JWK)", draft-ietf-jose-json-web-key (work in progress), October 2013.
- [JWS] Jones, M., Bradley, J., and N. Sakimura, "JSON Web Signature (JWS)", draft-ietf-jose-json-web-signature (work in progress), October 2013.
- [NIST.800-38A] National Institute of Standards and Technology (NIST), "Recommendation for Block Cipher Modes of Operation", NIST PUB 800-38A, December 2001.
- [NIST.800-38D] National Institute of Standards and Technology (NIST), "Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC", NIST PUB 800-38D, December 2001.
- [NIST.800-56A] National Institute of Standards and Technology (NIST), "Recommendation for Pair-Wise Key Establishment Schemes Using Discrete Logarithm Cryptography", NIST Special Publication 800-56A, Revision 2, May 2013.
- [RFC2104] Krawczyk, H., Bellare, M., and R. Canetti, "HMAC: Keyed-Hashing for Message Authentication", RFC 2104, February 1997.
- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.
- [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.

- [RFC4627] Crockford, D., "The application/json Media Type for JavaScript Object Notation (JSON)", RFC 4627, July 2006.
- [RFC4868] Kelly, S. and S. Frankel, "Using HMAC-SHA-256, HMAC-SHA-384, and HMAC-SHA-512 with IPsec", RFC 4868, May 2007.
- [RFC5116] McGrew, D., "An Interface and Algorithms for Authenticated Encryption", RFC 5116, January 2008.
- [RFC5226] Narten, T. and H. Alvestrand, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 5226, May 2008.
- [RFC6090] McGrew, D., Igoe, K., and M. Salter, "Fundamental Elliptic Curve Cryptography Algorithms", RFC 6090, February 2011.
- [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.

9.2. 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-xmlsig-core2-20120124]
Eastlake, D., Reagle, J., Yiu, K., Solo, D., Datta, P., Hirsch, F., Cantor, S., and T. Roessler, "XML Signature Syntax and Processing Version 2.0", World Wide Web Consortium CR CR-xmlsig-core2-20120124, January 2012, <<http://www.w3.org/TR/2012/CR-xmlsig-core2-20120124>>.
- [W3C.CR-xmlenc-core1-20120313]
Eastlake, D., Reagle, J., Roessler, T., and F. Hirsch, "XML Encryption Syntax and Processing Version 1.1", World Wide Web Consortium CR CR-xmlenc-core1-20120313, March 2012, <<http://www.w3.org/TR/2012/CR-xmlenc-core1-20120313>>.
- [W3C.REC-xmlenc-core-20021210]
Eastlake, D. and J. Reagle, "XML Encryption Syntax and Processing", World Wide Web Consortium Recommendation REC-xmlenc-core-20021210, December 2002, <<http://www.w3.org/TR/2002/REC-xmlenc-core-20021210>>.

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

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

JWS	XML DSIG	JCA	OID
HS256	http://www.w3.org/2001/04/xmlenc#sha256	HmacSHA256	1.2.840.11354.9.2.9
HS384	http://www.w3.org/2001/04/xmlenc#sha384	HmacSHA384	1.2.840.11354.9.2.10
HS512	http://www.w3.org/2001/04/xmlenc#sha512	HmacSHA512	1.2.840.11354.9.2.11
RS256	http://www.w3.org/2001/04/xmlenc#rsa-sha256	SHA256withRSA	1.2.840.11354.9.1.1.11
RS384	http://www.w3.org/2001/04/xmlenc#rsa-sha384	SHA384withRSA	1.2.840.11354.9.1.1.12
RS512	http://www.w3.org/2001/04/xmlenc#rsa-sha512	SHA512withRSA	1.2.840.11354.9.1.1.13
ES256	http://www.w3.org/2001/04/xmlenc#ecdsa-sha256	SHA256withECDSA	1.2.840.10045.4.3.2
ES384	http://www.w3.org/2001/04/xmlenc#ecdsa-sha384	SHA384withECDSA	1.2.840.10045.4.3.3
ES512	http://www.w3.org/2001/04/xmlenc#ecdsa-sha512	SHA512withECDSA	1.2.840.10045.4.3.4
PS256	http://www.w3.org/2007/05/xmlenc#sha256-rsa-MGF1		1.2.840.11354.9.1.1.10
PS384	http://www.w3.org/2007/05/xmlenc#sha384-rsa-MGF1		1.2.840.11354.9.1.1.10
PS512	http://www.w3.org/2007/05/xmlenc#sha512-rsa-MGF1		1.2.840.11354.9.1.1.10

Appendix B. Encryption Algorithm Identifier Cross-Reference

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

[JCA] for more information about the names defined by those documents.

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

JWE	XML ENC	JCA	OID
RSA1_5	http://www.w3.org/2001/04/xmlenc#rsa-1_5	RSA/ECB/PKCS1Padding	1.2.840.113549.1.1.1
RSA-OAEP	http://www.w3.org/2001/04/xmlenc#rsa-oaep-mgf1p	RSA/ECB/OAEPWithSHA-1AndMGF1Padding	1.2.840.113549.1.1.7
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		2.16.840.1.101.3.4.1.5
A192KW	http://www.w3.org/2001/04/xmlenc#kw-aes192		2.16.840.1.101.3.4.1.25
A256KW	http://www.w3.org/2001/04/xmlenc#kw-aes256		2.16.840.1.101.3.4.1.45
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 C. Test Cases for AES_CBC_HMAC_SHA2 Algorithms

The following test cases can be used to validate implementations of the AES_CBC_HMAC_SHA2 algorithms defined in Section 4.10. They are also intended to correspond to test cases that may appear in a future version of [I-D.mcgregre-aead-aes-cbc-hmac-sha2], demonstrating that the cryptographic computations performed are the same.

The variable names are those defined in Section 4.10. All values are hexadecimal.

C.1. Test Cases for AES_128_CBC_HMAC_SHA_256

AES_128_CBC_HMAC_SHA_256

```

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

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

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

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

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

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

AL =      00 00 00 00 00 00 01 50

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

```

C.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

```


C.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 D. 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 4.7. 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 128 bit keys ("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": "weNJy2HscCSM6AEDTDg04biOvhFhyWvOHQfeF_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_xSffzlPWrrHEVI30DHM_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].

PartyUIInfo This is set to the octets representing the 32 bit big endian value 5 - [0, 0, 0, 5] - the number of octets in the PartyUIInfo 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 PartyUIInfo 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 E. 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.mcgregw-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.mcgregw-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, 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 F. Document History

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

-17

- o 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.
- o Removed processing steps in algorithm definitions that duplicated processing steps in JWS or JWE, addressing issue #56.
- o Replaced verbose repetitive phrases such as "base64url encode the octets of the UTF-8 representation of X" with mathematical notation such as "BASE64URL(UTF8(X))".
- o 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.
- o Changes to address minor issue #53.

-16

- o Added a DataLen prefix to the AlgorithmID value in the Concat KDF computation.
- o Added OIDs for encryption algorithms, additional signature algorithm OIDs, and additional XML DSIG/ENC URIs in the algorithm cross-reference tables.
- o Changes to address editorial and minor issues #28, #36, #39, #52, #53, #55, #127, #128, #136, #137, #141, #150, #151, #152, and #155.

-15

- o Changed statements about rejecting JWSs to statements about validation failing, addressing issue #35.
- o Stated that changes of implementation requirements are only permitted on a Specification Required basis, addressing issue #38.
- o Made "oct" a required key type, addressing issue #40.
- o Updated the example ECDH-ES key agreement values.
- o 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

- o Removed "PBKDF2" key type and added "p2s" and "p2c" header parameters for use with the PBES2 algorithms.
- o Made the RSA private key parameters that are there to enable optimizations be RECOMMENDED rather than REQUIRED.
- o Added algorithm identifiers for AES algorithms using 192 bit keys and for RSASSA-PSS using HMAC SHA-384.
- o Added security considerations about key lifetimes, addressing issue #18.
- o Added an example ECDH-ES key agreement computation.

-13

- o Added key encryption with AES GCM as specified in draft-jones-jose-aes-gcm-key-wrap-01, addressing issue #13.
- o 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.
- o Added password-based key encryption as specified in draft-miller-jose-jwe-protected-jwk-02.

-12

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

- o Restored the "apv" (agreement PartyVInfo) parameter.
- o Moved the "epk", "apu", and "apv" Header Parameter definitions to be with the algorithm descriptions that use them.
- o Changed terminology from "block encryption" to "content encryption".

-11

- o Removed the Encrypted Key value from the AAD computation since it is already effectively integrity protected by the encryption process. The AAD value now only contains the representation of the JWE Encrypted Header.
- o Removed "apv" (agreement PartyVInfo) since it is no longer used.
- o Added more information about the use of PartyUInfo during key agreement.
- o Use the keydatalen as the SuppPubInfo value for the Concat KDF when doing key agreement, as RFC 2631 does.
- o Added algorithm identifiers for RSASSA-PSS with SHA-256 and SHA-512.
- o Added a Parameter Information Class value to the JSON Web Key Parameters registry, which registers whether the parameter conveys public or private information.

-10

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

-09

- o Expanded the scope of the JWK parameters to include private and symmetric key representations, as specified by draft-jones-jose-json-private-and-symmetric-key-00.
- o Changed term "JWS Secured Input" to "JWS Signing Input".
- o Changed from using the term "byte" to "octet" when referring to 8 bit values.

- o Specified that AES Key Wrap uses the default initial value specified in Section 2.2.3.1 of RFC 3394. This addressed issue #19.
- o Added Key Management Mode definitions to terminology section and used the defined terms to provide clearer key management instructions. This addressed issue #5.
- o Replaced "A128CBC+HS256" and "A256CBC+HS512" with "A128CBC-HS256" and "A256CBC-HS512". The new algorithms perform the same cryptographic computations as [I-D.mcgrewe-aead-aes-cbc-hmac-sha2], but with the Initialization Vector and Authentication Tag values remaining separate from the Ciphertext value in the output representation. Also deleted the header parameters "epu" (encryption PartyUInfo) and "epv" (encryption PartyVInfo), since they are no longer used.
- o Changed from using the term "Integrity Value" to "Authentication Tag".

-08

- o Changed the name of the JWK key type parameter from "alg" to "kty".
- o Replaced uses of the term "AEAD" with "Authenticated Encryption", since the term AEAD in the RFC 5116 sense implied the use of a particular data representation, rather than just referring to the class of algorithms that perform authenticated encryption with associated data.
- o Applied editorial improvements suggested by Jeff Hodges. Many of these simplified the terminology used.
- o Added seriesInfo information to Internet Draft references.

-07

- o Added a data length prefix to PartyUInfo and PartyVInfo values.
- o Changed the name of the JWK RSA modulus parameter from "mod" to "n" and the name of the JWK RSA exponent parameter from "xpo" to "e", so that the identifiers are the same as those used in RFC 3447.
- o Made several local editorial changes to clean up loose ends left over from the decision to only support block encryption methods providing integrity.

-06

- o Removed the "int" and "kdf" parameters and defined the new composite Authenticated Encryption algorithms "A128CBC+HS256" and "A256CBC+HS512" to replace the former uses of AES CBC, which required the use of separate integrity and key derivation functions.
- o Included additional values in the Concat KDF calculation -- the desired output size and the algorithm value, and optionally PartyUInfo and PartyVInfo values. Added the optional header parameters "apu" (agreement PartyUInfo), "apv" (agreement PartyVInfo), "epu" (encryption PartyUInfo), and "epv" (encryption PartyVInfo).
- o Changed the name of the JWK RSA exponent parameter from "exp" to "xpo" so as to allow the potential use of the name "exp" for a future extension that might define an expiration parameter for keys. (The "exp" name is already used for this purpose in the JWT specification.)
- o Applied changes made by the RFC Editor to RFC 6749's registry language to this specification.

-05

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

-04

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

-03

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

-02

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

-01

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

- o Added Advanced Encryption Standard (AES) Key Wrap Algorithm using 512 bit keys ("A512KW").
- o Added text "Alternatively, the Encoded JWS Signature MAY be base64url decoded to produce the JWS Signature and this value can be compared with the computed HMAC value, as this comparison produces the same result as comparing the encoded values".
- o Corrected the Magic Signatures reference.
- o Made other editorial improvements suggested by JOSE working group participants.

-00

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

Author's Address

Michael B. Jones
Microsoft

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

JOSE Working Group
Internet-Draft
Intended status: Standards Track
Expires: April 10, 2014

M. Jones
Microsoft
E. Rescorla
RTFM
J. Hildebrand
Cisco
October 7, 2013

JSON Web Encryption (JWE)
draft-ietf-jose-json-web-encryption-17

Abstract

JSON Web Encryption (JWE) represents encrypted content using JavaScript Object Notation (JSON) based data structures. Cryptographic algorithms and identifiers for use with this specification are described in the separate JSON Web Algorithms (JWA) specification and IANA registries defined by that specification. Related digital signature and MAC capabilities are described in the separate JSON Web Signature (JWS) specification.

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 April 10, 2014.

Copyright Notice

Copyright (c) 2013 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<http://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents

carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction	4
1.1. Notational Conventions	4
2. Terminology	5
3. JSON Web Encryption (JWE) Overview	7
3.1. Example JWE	9
4. JWE Header	11
4.1. Registered Header Parameter Names	11
4.1.1. "alg" (Algorithm) Header Parameter	11
4.1.2. "enc" (Encryption Method) Header Parameter	12
4.1.3. "zip" (Compression Algorithm) Header Parameter	12
4.1.4. "jku" (JWK Set URL) Header Parameter	12
4.1.5. "jwk" (JSON Web Key) Header Parameter	13
4.1.6. "x5u" (X.509 URL) Header Parameter	13
4.1.7. "x5t" (X.509 Certificate SHA-1 Thumbprint) Header Parameter	13
4.1.8. "x5c" (X.509 Certificate Chain) Header Parameter	13
4.1.9. "kid" (Key ID) Header Parameter	13
4.1.10. "typ" (Type) Header Parameter	13
4.1.11. "cty" (Content Type) Header Parameter	14
4.1.12. "crit" (Critical) Header Parameter	14
4.2. Public Header Parameter Names	14
4.3. Private Header Parameter Names	14
5. Producing and Consuming JWES	14
5.1. Message Encryption	14
5.2. Message Decryption	16
5.3. String Comparison Rules	19
6. Key Identification	19
7. Serializations	19
7.1. JWE Compact Serialization	19
7.2. JWE JSON Serialization	19
8. Distinguishing Between JWS and JWE Objects	22
9. IANA Considerations	23
9.1. JWE Header Parameter Names Registration	23
9.1.1. Registry Contents	23
10. Security Considerations	25
11. References	25
11.1. Normative References	25
11.2. Informative References	26
Appendix A. JWE Examples	27

A.1.	Example JWE using RSAES OAEP and AES GCM	27
A.1.1.	JWE Header	27
A.1.2.	Content Encryption Key (CEK)	27
A.1.3.	Key Encryption	28
A.1.4.	Initialization Vector	29
A.1.5.	Additional Authenticated Data	29
A.1.6.	Content Encryption	29
A.1.7.	Complete Representation	30
A.1.8.	Validation	30
A.2.	Example JWE using RSAES-PKCS1-V1_5 and AES_128_CBC_HMAC_SHA_256	30
A.2.1.	JWE Header	31
A.2.2.	Content Encryption Key (CEK)	31
A.2.3.	Key Encryption	31
A.2.4.	Initialization Vector	33
A.2.5.	Additional Authenticated Data	33
A.2.6.	Content Encryption	33
A.2.7.	Complete Representation	34
A.2.8.	Validation	34
A.3.	Example JWE using AES Key Wrap and AES_128_CBC_HMAC_SHA_256	34
A.3.1.	JWE Header	34
A.3.2.	Content Encryption Key (CEK)	35
A.3.3.	Key Encryption	35
A.3.4.	Initialization Vector	36
A.3.5.	Additional Authenticated Data	36
A.3.6.	Content Encryption	36
A.3.7.	Complete Representation	37
A.3.8.	Validation	37
A.4.	Example JWE using JWE JSON Serialization	37
A.4.1.	JWE Per-Recipient Unprotected Headers	38
A.4.2.	JWE Protected Header	38
A.4.3.	JWE Unprotected Header	38
A.4.4.	Complete JWE Header Values	38
A.4.5.	Additional Authenticated Data	39
A.4.6.	Content Encryption	39
A.4.7.	Complete JWE JSON Serialization Representation	39
Appendix B.	Example AES_128_CBC_HMAC_SHA_256 Computation	40
B.1.	Extract MAC_KEY and ENC_KEY from Key	40
B.2.	Encrypt Plaintext to Create Ciphertext	41
B.3.	64 Bit Big Endian Representation of AAD Length	41
B.4.	Initialization Vector Value	41
B.5.	Create Input to HMAC Computation	42
B.6.	Compute HMAC Value	42
B.7.	Truncate HMAC Value to Create Authentication Tag	42
Appendix C.	Acknowledgements	42
Appendix D.	Document History	43
Authors' Addresses	51

1. Introduction

JSON Web Encryption (JWE) represents encrypted content using JavaScript Object Notation (JSON) [RFC4627] based data structures. The JWE cryptographic mechanisms encrypt and provide integrity protection for an arbitrary sequence of octets.

Two closely related serializations for JWE objects are defined. The JWE Compact Serialization is a compact, URL-safe representation intended for space constrained environments such as HTTP Authorization headers and URI query parameters. The JWE JSON Serialization represents JWE objects as JSON objects and enables the same content to be encrypted to multiple parties. Both share the same cryptographic underpinnings.

Cryptographic algorithms and identifiers for use with this specification are described in the separate JSON Web Algorithms (JWA) [JWA] specification and IANA registries defined by that specification. Related digital signature and MAC capabilities are described in the separate JSON Web Signature (JWS) [JWS] specification.

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)", "JSON Text Object", "Base64url Encoding", "Collision Resistant Name", and "StringOrURI".

These terms are defined for use by this specification:

JSON Web Encryption (JWE) A data structure representing an encrypted and integrity protected message.

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

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

Ciphertext An encrypted representation of the Plaintext.

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

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

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

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

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

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

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

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

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

Header Parameter A name/value pair that is member of the JWE Header.

JWE Compact Serialization A representation of the JWE as the string
BASE64URL(UTF8(JWE Protected Header)) || '.' || BASE64URL(JWE Encrypted Key) || '.' || BASE64URL(JWE Initialization Vector) || '.' || BASE64URL(JWE Ciphertext) || '.' || BASE64URL(JWE Authentication Tag). This representation is compact and URL-safe.

JWE JSON Serialization A representation of the JWE as a JSON structure. Unlike the JWE Compact Serialization, the JWE JSON Serialization enables the same content to be encrypted to multiple parties. This representation is neither compact nor URL-safe.

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

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

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

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

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

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

3. JSON Web Encryption (JWE) Overview

JWE represents encrypted content using JSON data structures and base64url encoding. A JWE represents these logical values:

JWE Header JSON object containing the parameters describing the cryptographic operations and parameters employed. The JWE Header members are the union of the members of the JWE Protected Header, the JWE Shared Unprotected Header, and the JWE Per-Recipient Unprotected Header, as described below.

JWE Encrypted Key Encrypted Content Encryption Key (CEK) value.

JWE Initialization Vector Initialization Vector value used when encrypting the plaintext.

JWE Ciphertext Ciphertext value resulting from authenticated encryption of the plaintext.

JWE Authentication Tag Authentication Tag value resulting from authenticated encryption of the plaintext with additional associated data.

JWE AAD Additional value to be integrity protected by the authenticated encryption operation.

The JWE Header represents the combination of these logical values:

JWE Protected Header JSON object containing some of the parameters describing the cryptographic operations and parameters employed. These parameters apply to all recipients of the JWE. This value is integrity protected in the digital signature or MAC calculation of the JWE Signature.

JWE Shared Unprotected Header JSON object containing some of the parameters describing the cryptographic operations and parameters employed. These parameters apply to all recipients of the JWE. This value is not integrity protected in the authenticated encryption operation.

JWE Per-Recipient Unprotected Header JSON object containing some of the parameters describing the cryptographic operations and parameters employed. These parameters apply to a single recipient of the JWE. This value is not integrity protected in the authenticated encryption operation.

This document defines two serializations for JWE objects: a compact, URL-safe serialization called the JWE Compact Serialization and a JSON serialization called the JWE JSON Serialization. In both serializations, the JWE Protected Header, JWE Encrypted Key, JWE Initialization Vector, JWE Ciphertext, and JWE Authentication Tag are base64url encoded for transmission, since JSON lacks a way to directly represent octet sequences. When present, the JWE AAD is also base64url encoded for transmission.

In the JWE Compact Serialization, no JWE Shared Unprotected Header or JWE Per-Recipient Unprotected Header are used. In this case, the JWE Header and the JWE Protected Header are the same.

In the JWE Compact Serialization, a JWE object is represented as the combination of these five string values,

BASE64URL(UTF8(JWE Protected Header)),
BASE64URL(JWE Encrypted Key),
BASE64URL(JWE Initialization Vector),
BASE64URL(JWE Ciphertext), and
BASE64URL(JWE Authentication Tag),

concatenated in that order, with the five strings being separated by four period ('.') characters.

In the JWE JSON Serialization, one or more of the JWE Protected Header, JWE Shared Unprotected Header, and JWE Per-Recipient Unprotected Header MUST be present. In this case, the members of the JWE Header are the combination of the members of the JWE Protected Header, JWE Shared Unprotected Header, and JWE Per-Recipient Unprotected Header values that are present.

In the JWE JSON Serialization, a JWE object is represented as the combination of these eight values,

- BASE64URL(UTF8(JWE Protected Header)),
- JWE Shared Unprotected Header,
- JWE Per-Recipient Unprotected Header,
- BASE64URL(JWE Encrypted Key),
- BASE64URL(JWE Initialization Vector),
- BASE64URL(JWE Ciphertext),
- BASE64URL(JWE Authentication Tag), and
- BASE64URL(JWE AAD),

with the six base64url encoding result strings and the two unprotected JSON object values being represented as members within a JSON object. The inclusion of some of these values is OPTIONAL. The JWE JSON Serialization can also encrypt the plaintext to multiple recipients. See Section 7.2 for more information about the JWE JSON Serialization.

JWE utilizes authenticated encryption to ensure the confidentiality and integrity of the Plaintext and the integrity of the JWE Protected Header and the JWE AAD.

3.1. Example JWE

This example encrypts the plaintext "The true sign of intelligence is not knowledge but imagination." to the recipient using RSAES OAEP for key encryption and AES GCM for content encryption.

The following example JWE Protected Header declares that:

- o the Content Encryption Key is encrypted to the recipient using the RSAES OAEP algorithm to produce the JWE Encrypted Key and
- o the Plaintext is encrypted using the AES GCM algorithm with a 256 bit key to produce the Ciphertext.

```
{"alg":"RSA-OAEP","enc":"A256GCM"}
```

Encoding this JWE Protected Header as BASE64URL(UTF8(JWE Protected Header)) gives this value:

```
eyJhbGciOiJSU0EtT0FFUCIsImVuYyI6IkJEYNTZHQ00ifQ
```

The remaining steps to finish creating this JWE are:

- o Generate a random Content Encryption Key (CEK).

- o Encrypt the CEK with the recipient's public key using the RSAES OAEP algorithm to produce the JWE Encrypted Key.
- o Base64url encode the JWE Encrypted Key.
- o Generate a random JWE Initialization Vector.
- o Base64url encode the JWE Initialization Vector.
- o Let the Additional Authenticated Data encryption parameter be ASCII(BASE64URL(UTF8(JWE Protected Header))).
- o Encrypt the Plaintext with AES GCM using the CEK as the encryption key, the JWE Initialization Vector, and the Additional Authenticated Data value, requesting a 128 bit Authentication Tag output.
- o Base64url encode the Ciphertext.
- o Base64url encode the Authentication Tag.
- o Assemble the final representation: The Compact Serialization of this result is the string BASE64URL(UTF8(JWE Protected Header)) || '.' || BASE64URL(JWE Encrypted Key) || '.' || BASE64URL(JWE Initialization Vector) || '.' || BASE64URL(JWE Ciphertext) || '.' || BASE64URL(JWE Authentication Tag).

The final result in this example (with line breaks for display purposes only) is:

```
eyJhbGciOiJSU0EtT0FFUCIsImVuYyI6IkJEYNTZHQ00ifQ.
OKOawDol3gRp2ojaHV7LFpZcgV7T6DVZKTyKOMTYUmKoTCVJRgckCL9kiMT03JGe
ipsEdY3mx_etLbbWSrFr05kLzcSr4qKAq7YN7e9jwQRb23nfa6c9d-StnImGyFDb
Sv04uVuxIp5ZmslgNxKKK2Da14B8S4rzVRltdYwam_lDp5XnZAYpQdb76FdIKLaV
mqgfwX7XWRxv2322i-vDxRfqNzo_tETKzpVLzfiwQyeyPGLBIO56YJ7eObdv0je8
1860ppamavo35UgoRdbYaBcoh9QcfylQr66oc6vFWXRcZ_ZT2LawVCWTIy3brGPi
6UklfCpIMfIjf7iGdXKHZg.
48Vl_ALb6US04U3b.
5eym8TW_c8SuK0ltJ3rpYIzOeDQz7TALvtu6UG9oMo4vpzs9tX_EFShS8iB7j6ji
SdiwkIr3ajwQzaBtQD_A.
XFBOMYUZodetZdvTiFvSkQ
```

See Appendix A.1 for the complete details of computing this JWE. See Appendix A for additional examples.

4. JWE Header

The members of the JSON object(s) representing the JWE Header describe the encryption applied to the Plaintext and optionally additional properties of the JWE. The Header Parameter names within the JWE Header MUST be unique; recipients MUST either reject JWEs with duplicate Header Parameter names or use a JSON parser that returns only the lexically last duplicate member name, as specified in Section 15.12 (The JSON Object) of ECMA Script 5.1 [ECMA Script].

Implementations are required to understand the specific Header Parameters defined by this specification that are designated as "MUST be understood" and process them in the manner defined in this specification. All other Header Parameters defined by this specification that are not so designated MUST be ignored when not understood. Unless listed as a critical Header Parameter, per Section 4.1.12, all Header Parameters not defined by this specification MUST be ignored when not understood.

There are three classes of Header Parameter names: Registered Header Parameter names, Public Header Parameter names, and Private Header Parameter names.

4.1. Registered Header Parameter Names

The following Header Parameter names are registered in the IANA JSON Web Signature and Encryption Header Parameters registry defined in [JWS], with meanings as defined below.

As indicated by the common registry, JWSs and JWEs share a common Header Parameter space; when a parameter is used by both specifications, its usage must be compatible between the specifications.

4.1.1. "alg" (Algorithm) Header Parameter

This parameter has the same meaning, syntax, and processing rules as the "alg" Header Parameter defined in Section 4.1.1 of [JWS], except that the Header Parameter identifies the cryptographic algorithm used to encrypt or determine the value of the Content Encryption Key (CEK). The encrypted content is not usable if the "alg" value does not represent a supported algorithm, or if the recipient does not have a key that can be used with that algorithm.

A list of defined "alg" values for this use can be found in the IANA JSON Web Signature and Encryption Algorithms registry defined in [JWA]; the initial contents of this registry are the values defined in Section 4.1 of the JSON Web Algorithms (JWA) [JWA] specification.

4.1.2. "enc" (Encryption Method) Header Parameter

The "enc" (encryption method) Header Parameter identifies the content encryption algorithm used to encrypt the Plaintext to produce the Ciphertext. This algorithm MUST be an AEAD algorithm with a specified key length. The recipient MUST reject the JWE if the "enc" value does not represent a supported algorithm. "enc" values SHOULD either be registered in the IANA JSON Web Signature and Encryption Algorithms registry defined in [JWA] or be a value that contains a Collision Resistant Name. The "enc" value is a case sensitive string containing a StringOrURI value. Use of this Header Parameter is REQUIRED. This Header Parameter MUST be understood and processed by implementations.

A list of defined "enc" values for this use can be found in the IANA JSON Web Signature and Encryption Algorithms registry defined in [JWA]; the initial contents of this registry are the values defined in Section 4.2 of the JSON Web Algorithms (JWA) [JWA] specification.

4.1.3. "zip" (Compression Algorithm) Header Parameter

The "zip" (compression algorithm) applied to the Plaintext before encryption, if any. The "zip" value defined by this specification is:

- o "DEF" - Compression with the DEFLATE [RFC1951] algorithm

Other values MAY be used. Compression algorithm values can be registered in the IANA JSON Web Encryption Compression Algorithm registry defined in [JWA]. The "zip" value is a case sensitive string. If no "zip" parameter is present, no compression is applied to the Plaintext before encryption. This Header Parameter MUST be integrity protected, and therefore MUST occur only within the JWE Protected Header, when used. Use of this Header Parameter is OPTIONAL. This Header Parameter MUST be understood and processed by implementations.

4.1.4. "jku" (JWK Set URL) Header Parameter

This parameter has the same meaning, syntax, and processing rules as the "jku" Header Parameter defined in Section 4.1.2 of [JWS], except that the JWK Set resource contains the public key to which the JWE was encrypted; this can be used to determine the private key needed to decrypt the JWE.

4.1.5. "jwk" (JSON Web Key) Header Parameter

This parameter has the same meaning, syntax, and processing rules as the "jwk" Header Parameter defined in Section 4.1.3 of [JWS], except that the key is the public key to which the JWE was encrypted; this can be used to determine the private key needed to decrypt the JWE.

4.1.6. "x5u" (X.509 URL) Header Parameter

This parameter has the same meaning, syntax, and processing rules as the "x5u" Header Parameter defined in Section 4.1.4 of [JWS], except that the X.509 public key certificate or certificate chain [RFC5280] contains the public key to which the JWE was encrypted; this can be used to determine the private key needed to decrypt the JWE.

4.1.7. "x5t" (X.509 Certificate SHA-1 Thumbprint) Header Parameter

This parameter has the same meaning, syntax, and processing rules as the "x5t" Header Parameter defined in Section 4.1.5 of [JWS], except that certificate referenced by the thumbprint contains the public key to which the JWE was encrypted; this can be used to determine the private key needed to decrypt the JWE.

4.1.8. "x5c" (X.509 Certificate Chain) Header Parameter

This parameter has the same meaning, syntax, and processing rules as the "x5c" Header Parameter defined in Section 4.1.6 of [JWS], except that the X.509 public key certificate or certificate chain [RFC5280] contains the public key to which the JWE was encrypted; this can be used to determine the private key needed to decrypt the JWE.

See Appendix B of [JWS] for an example "x5c" value.

4.1.9. "kid" (Key ID) Header Parameter

This parameter has the same meaning, syntax, and processing rules as the "kid" Header Parameter defined in Section 4.1.7 of [JWS], except that the key hint references the public key to which the JWE was encrypted; this can be used to determine the private key needed to decrypt the JWE. This parameter allows originators to explicitly signal a change of key to JWE recipients.

4.1.10. "typ" (Type) Header Parameter

This parameter has the same meaning, syntax, and processing rules as the "typ" Header Parameter defined in Section 4.1.8 of [JWS], except that the type is of this complete JWE object.

4.1.11. "cty" (Content Type) Header Parameter

This parameter has the same meaning, syntax, and processing rules as the "cty" Header Parameter defined in Section 4.1.9 of [JWS], except that the type is of the secured content (the payload).

4.1.12. "crit" (Critical) Header Parameter

This parameter has the same meaning, syntax, and processing rules as the "typ" Header Parameter defined in Section 4.1.8 of [JWS], except that the Header Parameters referenced are the JWE Header Parameters.

4.2. Public Header Parameter Names

Additional Header Parameter names can be defined by those using JWEs. However, in order to prevent collisions, any new Header Parameter name SHOULD either be registered in the IANA JSON Web Signature and Encryption Header Parameters registry defined in [JWS] or be a Public Name: a value that contains a Collision Resistant Name. In each case, the definer of the name or value needs to take reasonable precautions to make sure they are in control of the part of the namespace they use to define the Header Parameter name.

New Header Parameters should be introduced sparingly, as they can result in non-interoperable JWEs.

4.3. Private Header Parameter Names

A producer and consumer of a JWE may agree to use Header Parameter names that are Private Names: names that are not Registered Header Parameter names Section 4.1 or Public Header Parameter names Section 4.2. Unlike Public Header Parameter names, Private Header Parameter names are subject to collision and should be used with caution.

5. Producing and Consuming JWEs

5.1. Message Encryption

The message encryption process is as follows. The order of the steps is not significant in cases where there are no dependencies between the inputs and outputs of the steps.

1. Determine the Key Management Mode employed by the algorithm used to determine the Content Encryption Key (CEK) value. (This is the algorithm recorded in the "alg" (algorithm) Header Parameter of the resulting JWE.)

2. When Key Wrapping, Key Encryption, or Key Agreement with Key Wrapping are employed, generate a random Content Encryption Key (CEK) value. See RFC 4086 [RFC4086] for considerations on generating random values. The CEK MUST have a length equal to that required for the content encryption algorithm.
3. When Direct Key Agreement or Key Agreement with Key Wrapping are employed, use the key agreement algorithm to compute the value of the agreed upon key. When Direct Key Agreement is employed, let the Content Encryption Key (CEK) be the agreed upon key. When Key Agreement with Key Wrapping is employed, the agreed upon key will be used to wrap the CEK.
4. When Key Wrapping, Key Encryption, or Key Agreement with Key Wrapping are employed, encrypt the CEK to the recipient and let the result be the JWE Encrypted Key.
5. Otherwise, when Direct Key Agreement or Direct Encryption are employed, let the JWE Encrypted Key be the empty octet sequence.
6. When Direct Encryption is employed, let the Content Encryption Key (CEK) be the shared symmetric key.
7. Compute the encoded key value `BASE64URL(JWE Encrypted Key)`.
8. If the JWE JSON Serialization is being used, repeat this process for each recipient.
9. Generate a random JWE Initialization Vector of the correct size for the content encryption algorithm (if required for the algorithm); otherwise, let the JWE Initialization Vector be the empty octet sequence.
10. Compute the encoded initialization vector value `BASE64URL(JWE Initialization Vector)`.
11. Compress the Plaintext if a "zip" parameter was included.
12. Serialize the (compressed) Plaintext into an octet sequence M.
13. Create a JWE Header containing the encryption parameters used. Note that white space is explicitly allowed in the representation and no canonicalization need be performed before encoding.
14. Compute the encoded header value `BASE64URL(UTF8(JWE Protected Header))`. If the JWE Protected Header is not present (which can only happen when using the JWE JSON Serialization and no

"protected" member is present), let this value be the empty string.

15. Let the Additional Authenticated Data encryption parameter be ASCII(BASE64URL(UTF8(JWE Protected Header))). However if a JWE AAD value is present when using the JWE JSON Serialization, instead let the Additional Authenticated Data encryption parameter be ASCII(BASE64URL(UTF8(JWE Protected Header)) || '.' || BASE64URL(JWE AAD)).
16. Encrypt M using the CEK, the JWE Initialization Vector, and the Additional Authenticated Data value using the specified content encryption algorithm to create the JWE Ciphertext value and the JWE Authentication Tag (which is the Authentication Tag output from the encryption operation).
17. Compute the encoded ciphertext value BASE64URL(JWE Ciphertext).
18. Compute the encoded authentication tag value BASE64URL(JWE Authentication Tag).
19. The five encoded values are used in both the JWE Compact Serialization and the JWE JSON Serialization representations.
20. If a JWE AAD value is present, compute the encoded AAD value BASE64URL(JWE AAD).
21. Create the desired serialized output. The Compact Serialization of this result is the string BASE64URL(UTF8(JWE Protected Header)) || '.' || BASE64URL(JWE Encrypted Key) || '.' || BASE64URL(JWE Initialization Vector) || '.' || BASE64URL(JWE Ciphertext) || '.' || BASE64URL(JWE Authentication Tag). The JWE JSON Serialization is described in Section 7.2.

5.2. Message Decryption

The message decryption process is the reverse of the encryption process. The order of the steps is not significant in cases where there are no dependencies between the inputs and outputs of the steps. If any of these steps fails, the encrypted content cannot be validated.

It is an application decision which recipients' encrypted content must successfully validate for the JWE to be accepted. In some cases, encrypted content for all recipients must successfully validate or the JWE will be rejected. In other cases, only the encrypted content for a single recipient needs to be successfully validated. However, in all cases, the encrypted content for at least

one recipient MUST successfully validate or the JWE MUST be rejected.

1. Parse the JWE representation to extract the serialized values for the components of the JWE -- when using the JWE Compact Serialization, the base64url encoded representations of the JWE Protected Header, the JWE Encrypted Key, the JWE Initialization Vector, the JWE Ciphertext, and the JWE Authentication Tag, and when using the JWE JSON Serialization, also the base64url encoded representation of the JWE AAD and the unencoded JWE Shared Unprotected Header and JWE Per-Recipient Unprotected Header values. When using the JWE Compact Serialization, the JWE Protected Header, the JWE Encrypted Key, the JWE Initialization Vector, the JWE Ciphertext, and the JWE Authentication Tag are represented as base64url encoded values in that order, separated by four period ('.') characters. The JWE JSON Serialization is described in Section 7.2.
2. The encoded representations of the JWE Protected Header, the JWE Encrypted Key, the JWE Initialization Vector, the JWE Ciphertext, the JWE Authentication Tag, and the JWE AAD MUST be successfully base64url decoded following the restriction that no padding characters have been used.
3. The resulting UTF8 encoded JWE Protected Header MUST be a completely valid JSON object conforming to RFC 4627 [RFC4627].
4. If using the JWE Compact Serialization, let the JWE Header be the JWE Protected Header; otherwise, when using the JWE JSON Serialization, let the JWE Header be the union of the members of the JWE Protected Header, the JWE Shared Unprotected Header and the corresponding JWE Per-Recipient Unprotected Header, all of which must be completely valid JSON objects.
5. The resulting JWE Header MUST NOT contain duplicate Header Parameter names. When using the JWE JSON Serialization, this restriction includes that the same Header Parameter name also MUST NOT occur in distinct JSON Text Object values that together comprise the JWE Header.
6. The resulting JWE Header MUST be validated to only include parameters and values whose syntax and semantics are both understood and supported or that are specified as being ignored when not understood.
7. Determine the Key Management Mode employed by the algorithm specified by the "alg" (algorithm) Header Parameter.

8. Verify that the JWE uses a key known to the recipient.
9. When Direct Key Agreement or Key Agreement with Key Wrapping are employed, use the key agreement algorithm to compute the value of the agreed upon key. When Direct Key Agreement is employed, let the Content Encryption Key (CEK) be the agreed upon key. When Key Agreement with Key Wrapping is employed, the agreed upon key will be used to decrypt the JWE Encrypted Key.
10. When Key Wrapping, Key Encryption, or Key Agreement with Key Wrapping are employed, decrypt the JWE Encrypted Key to produce the Content Encryption Key (CEK). The CEK MUST have a length equal to that required for the content encryption algorithm. Note that when there are multiple recipients, each recipient will only be able to decrypt any JWE Encrypted Key values that were encrypted to a key in that recipient's possession. It is therefore normal to only be able to decrypt one of the per-recipient JWE Encrypted Key values to obtain the CEK value. To mitigate the attacks described in RFC 3218 [RFC3218], the recipient MUST NOT distinguish between format, padding, and length errors of encrypted keys. It is strongly recommended, in the event of receiving an improperly formatted key, that the receiver substitute a randomly generated CEK and proceed to the next step, to mitigate timing attacks.
11. Otherwise, when Direct Key Agreement or Direct Encryption are employed, verify that the JWE Encrypted Key value is empty octet sequence.
12. When Direct Encryption is employed, let the Content Encryption Key (CEK) be the shared symmetric key.
13. If the JWE JSON Serialization is being used, repeat this process for each recipient contained in the representation until the CEK value has been determined.
14. Let the Additional Authenticated Data encryption parameter be ASCII(BASE64URL(UTF8(JWE Protected Header))). However if a JWE AAD value is present when using the JWE JSON Serialization, instead let the Additional Authenticated Data encryption parameter be ASCII(BASE64URL(UTF8(JWE Protected Header)) || '.' || BASE64URL(JWE AAD)).
15. Decrypt the JWE Ciphertext using the CEK, the JWE Initialization Vector, the Additional Authenticated Data value, and the JWE Authentication Tag (which is the Authentication Tag input to the calculation) using the specified content encryption algorithm, returning the decrypted plaintext and verifying the JWE

Authentication Tag in the manner specified for the algorithm, rejecting the input without emitting any decrypted output if the JWE Authentication Tag is incorrect.

16. Uncompress the decrypted plaintext if a "zip" parameter was included.
17. Output the resulting Plaintext.

5.3. String Comparison Rules

The string comparison rules for this specification are the same as those defined in Section 5.3 of [JWS].

6. Key Identification

The key identification methods for this specification are the same as those defined in Section 6 of [JWS], except that the key being identified is the public key to which the JWE was encrypted.

7. Serializations

JWE objects use one of two serializations, the JWE Compact Serialization or the JWE JSON Serialization. For general-purpose implementations, both the JWE Compact Serialization and JWE JSON Serialization support for the single recipient case are mandatory to implement; support for multiple recipients is OPTIONAL. Special-purpose implementations are permitted to implement only a single serialization, if that meets the needs of the targeted use cases.

7.1. JWE Compact Serialization

The JWE Compact Serialization represents encrypted content as a compact URL-safe string. This string is `BASE64URL(UTF8(JWE Protected Header)) || '.' || BASE64URL(JWE Encrypted Key) || '.' || BASE64URL(JWE Initialization Vector) || '.' || BASE64URL(JWE Ciphertext) || '.' || BASE64URL(JWE Authentication Tag)`. Only one recipient is supported by the JWE Compact Serialization.

7.2. JWE JSON Serialization

The JWE JSON Serialization represents encrypted content as a JSON object. Unlike the JWE Compact Serialization, content using the JWE JSON Serialization can be encrypted to more than one recipient.

The representation is closely related to that used in the JWE Compact

Serialization, with the following differences for the JWE JSON Serialization:

- o Values in the JWE JSON Serialization are represented as members of a JSON object, rather than as base64url encoded strings separated by period ('.') characters. (However binary values and values that are integrity protected are still base64url encoded.)
- o The value `BASE64URL(UTF8(JWE Protected Header))`, if non-empty, is stored in the "protected" member.
- o The value `BASE64URL(UTF8(JWE Shared Unprotected Header))`, if non-empty, is stored in the "unprotected" member.
- o The value `BASE64URL(JWE Initialization Vector)`, if non-empty, is stored in the "iv" member.
- o The value `BASE64URL(JWE Ciphertext)` is stored in the "ciphertext" member.
- o The value `BASE64URL(JWE Authentication Tag)`, if non-empty, is stored in the "tag" member.
- o The JWE can be encrypted to multiple recipients, rather than just one. A JSON array in the "recipients" member is used to hold values that are specific to a particular recipient, with one array element per recipient represented. These array elements are JSON objects.
- o Each value `BASE64URL(JWE Encrypted Key)`, if non-empty, is stored in the "encrypted_key" member of a JSON object that is an element of the "recipients" array.
- o Some Header Parameter values, such as the "alg" value and parameters used for selecting keys, can also differ for different recipient computations. JWE Per-Recipient Unprotected Header values, if present, are stored in the "header" members of the same JSON objects that are elements of the "recipients" array.
- o Some Header Parameters, including the "alg" parameter, can be shared among all recipient computations. Header Parameters in the JWE Protected Header and JWE Shared Unprotected Header values are shared among all recipients.
- o Not all Header Parameters are integrity protected. The shared Header Parameters in the JWE Protected Header value member are integrity protected, and are base64url encoded for transmission. The per-recipient Header Parameters in the JWE Per-Recipient

Unprotected Header values and the shared Header Parameters in the JWE Shared Unprotected Header value are not integrity protected. These JSON Text Objects containing Header Parameters that are not integrity protected are not base64url encoded.

- o The Header Parameter values used when creating or validating per-recipient Ciphertext and Authentication Tag values are the union of the three sets of Header Parameter values that may be present: (1) the JWE Protected Header values represented in the "protected" member, (2) the JWE Shared Unprotected Header values represented in the "unprotected" member, and (3) the JWE Per-Recipient Unprotected Header values represented in the "header" member of the recipient's array element. The union of these sets of Header Parameters comprises the JWE Header. The Header Parameter names in the three locations MUST be disjoint.
- o A JWE AAD value can be included to supply a base64url encoded value to be integrity protected but not encrypted. (Note that this can also be achieved when using either serialization by including the AAD value as a protected Header Parameter value, but at the cost of the value being double base64url encoded.) If a JWE AAD value is present, the value `BASE64URL(JWE AAD)` is stored in the "aad" member.
- o The "recipients" array MUST always be present, even if the array elements contain only the empty JSON object "{}" (which can happen when all Header Parameter values are shared between all recipients and when no encrypted key is used, such as when doing Direct Encryption).

The syntax of a JWE using the JWE JSON Serialization is as follows:

```
{ "protected": "<integrity-protected shared header contents>",  
  "unprotected": "<non-integrity-protected shared header contents>",  
  "recipients": [  
    { "header": "<per-recipient unprotected header 1 contents>",  
      "encrypted_key": "<encrypted key 1 contents>" },  
    ...  
    { "header": "<per-recipient unprotected header N contents>",  
      "encrypted_key": "<encrypted key N contents>" } ],  
  "aad": "<additional authenticated data contents>",  
  "iv": "<initialization vector contents>",  
  "ciphertext": "<ciphertext contents>",  
  "tag": "<authentication tag contents>"  
}
```

Of these members, only the "ciphertext" member MUST be present. The "iv", "tag", and "encrypted_key" members MUST be present when

corresponding JWE Initialization Vector, JWE Authentication Tag, and JWE Encrypted Key values are non-empty. The "recipients" member MUST be present when any "header" or "encrypted_key" members are needed for recipients. At least one of the "header", "protected", and "unprotected" members MUST be present so that "alg" and "enc" Header Parameter values are conveyed for each recipient computation.

The contents of the JWE Encrypted Key, JWE Initialization Vector, JWE Ciphertext, and JWE Authentication Tag values are exactly as defined in the rest of this specification. They are interpreted and validated in the same manner, with each corresponding JWE Encrypted Key, JWE Initialization Vector, JWE Ciphertext, JWE Authentication Tag, and set of Header Parameter values being created and validated together. The JWE Header values used are the union of the Header Parameters in the JWE Protected Header, JWE Shared Unprotected Header, and corresponding JWE Per-Recipient Unprotected Header values, as described earlier.

Each JWE Encrypted Key value is computed using the parameters of the corresponding JWE Header value in the same manner as for the JWE Compact Serialization. This has the desirable property that each JWE Encrypted Key value in the "recipients" array is identical to the value that would have been computed for the same parameter in the JWE Compact Serialization. Likewise, the JWE Ciphertext and JWE Authentication Tag values match those produced for the JWE Compact Serialization, provided that the JWE Protected Header value (which represents the integrity-protected Header Parameter values) matches that used in the JWE Compact Serialization.

All recipients use the same JWE Protected Header, JWE Initialization Vector, JWE Ciphertext, and JWE Authentication Tag values, resulting in potentially significant space savings if the message is large. Therefore, all Header Parameters that specify the treatment of the Plaintext value MUST be the same for all recipients. This primarily means that the "enc" (encryption method) Header Parameter value in the JWE Header for each recipient and any parameters of that algorithm MUST be the same.

See Appendix A.4 for an example of computing a JWE using the JWE JSON Serialization.

8. Distinguishing Between JWS and JWE Objects

There are several ways of distinguishing whether an object is a JWS or JWE object. All these methods will yield the same result for all legal input values.

- o If the object is using the JWS Compact Serialization or the JWE Compact Serialization, the number of base64url encoded segments separated by period ('.') characters differs for JWSs and JWEs. JWSs have three segments separated by two period ('.') characters. JWEs have five segments separated by four period ('.') characters.
- o If the object is using the JWS JSON Serialization or the JWE JSON Serialization, the members used will be different. JWSs have a "signatures" member and JWEs do not. JWEs have a "recipients" member and JWSs do not.
- o A JWS Header can be distinguished from a JWE header by examining the "alg" (algorithm) Header Parameter value. If the value represents a digital signature or MAC algorithm, or is the value "none", it is for a JWS; if it represents a Key Encryption, Key Wrapping, Direct Key Agreement, Key Agreement with Key Wrapping, or Direct Encryption algorithm, it is for a JWE.
- o A JWS Header can also be distinguished from a JWE header by determining whether an "enc" (encryption method) member exists. If the "enc" member exists, it is a JWE; otherwise, it is a JWS.

9. IANA Considerations

9.1. JWE Header Parameter Names Registration

This specification registers the Header Parameter names defined in Section 4.1 in the IANA JSON Web Signature and Encryption Header Parameters registry defined in [JWS].

9.1.1. Registry Contents

- o Header Parameter Name: "alg"
- o Header Parameter Usage Location(s): JWE
- o Change Controller: IESG
- o Specification Document(s): Section 4.1.1 of [[this document]]
- o Header Parameter Name: "enc"
- o Header Parameter Usage Location(s): JWE
- o Change Controller: IESG
- o Specification Document(s): Section 4.1.2 of [[this document]]
- o Header Parameter Name: "zip"
- o Header Parameter Usage Location(s): JWE
- o Change Controller: IESG

- o Specification Document(s): Section 4.1.3 of [[this document]]
- o Header Parameter Name: "jku"
- o Header Parameter Usage Location(s): JWE
- o Change Controller: IESG
- o Specification Document(s): Section 4.1.4 of [[this document]]
- o Header Parameter Name: "jwk"
- o Header Parameter Usage Location(s): JWE
- o Change Controller: IESG
- o Specification document(s): Section 4.1.5 of [[this document]]
- o Header Parameter Name: "x5u"
- o Header Parameter Usage Location(s): JWE
- o Change Controller: IESG
- o Specification Document(s): Section 4.1.6 of [[this document]]
- o Header Parameter Name: "x5t"
- o Header Parameter Usage Location(s): JWE
- o Change Controller: IESG
- o Specification Document(s): Section 4.1.7 of [[this document]]
- o Header Parameter Name: "x5c"
- o Header Parameter Usage Location(s): JWE
- o Change Controller: IESG
- o Specification Document(s): Section 4.1.8 of [[this document]]
- o Header Parameter Name: "kid"
- o Header Parameter Usage Location(s): JWE
- o Change Controller: IESG
- o Specification Document(s): Section 4.1.9 of [[this document]]
- o Header Parameter Name: "typ"
- o Header Parameter Usage Location(s): JWE
- o Change Controller: IESG
- o Specification Document(s): Section 4.1.10 of [[this document]]
- o Header Parameter Name: "cty"
- o Header Parameter Usage Location(s): JWE
- o Change Controller: IESG
- o Specification Document(s): Section 4.1.11 of [[this document]]
- o Header Parameter Name: "crit"
- o Header Parameter Usage Location(s): JWE
- o Change Controller: IESG
- o Specification Document(s): Section 4.1.12 of [[this document]]

10. 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.

All the security considerations in the JWS specification also apply to this specification. Likewise, all the security considerations in XML Encryption 1.1 [W3C.CR-xmlenc-core1-20120313] also apply, other than those that are XML specific.

When decrypting, particular care must be taken not to allow the JWE recipient to be used as an oracle for decrypting messages. RFC 3218 [RFC3218] should be consulted for specific countermeasures to attacks on RSAES-PKCS1-V1_5. An attacker might modify the contents of the "alg" parameter from "RSA-OAEP" to "RSA1_5" in order to generate a formatting error that can be detected and used to recover the CEK even if RSAES OAEP was used to encrypt the CEK. It is therefore particularly important to report all formatting errors to the CEK, Additional Authenticated Data, or ciphertext as a single error when the encrypted content is rejected.

11. References

11.1. Normative References

- [ECMAScript] Ecma International, "ECMAScript Language Specification, 5.1 Edition", ECMA 262, June 2011.
- [JWA] Jones, M., "JSON Web Algorithms (JWA)", draft-ietf-jose-json-web-algorithms (work in progress), October 2013.
- [JWK] Jones, M., "JSON Web Key (JWK)", draft-ietf-jose-json-web-key (work in progress), October 2013.
- [JWS] Jones, M., Bradley, J., and N. Sakimura, "JSON Web Signature (JWS)", draft-ietf-jose-json-web-signature (work in progress), October 2013.
- [RFC1951] Deutsch, P., "DEFLATE Compressed Data Format Specification version 1.3", RFC 1951, May 1996.

- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.
- [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.
- [RFC4627] Crockford, D., "The application/json Media Type for JavaScript Object Notation (JSON)", RFC 4627, July 2006.
- [RFC5280] Cooper, D., Santesson, S., Farrell, S., Boeyen, S., Housley, R., and W. Polk, "Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile", RFC 5280, May 2008.
- [USASCII] American National Standards Institute, "Coded Character Set -- 7-bit American Standard Code for Information Interchange", ANSI X3.4, 1986.
- [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>>.

11.2. Informative References

- [I-D.mcgregw-aead-aes-cbc-hmac-sha2]
McGrew, D. and K. Paterson, "Authenticated Encryption with AES-CBC and HMAC-SHA",
draft-mcgregw-aead-aes-cbc-hmac-sha2-01 (work in progress),
October 2012.
- [I-D.rescorla-jsms]
Rescorla, E. and J. Hildebrand, "JavaScript Message Security Format", draft-rescorla-jsms-00 (work in progress), March 2011.
- [JSE] Bradley, J. and N. Sakimura (editor), "JSON Simple Encryption", September 2010.
- [RFC3218] Rescorla, E., "Preventing the Million Message Attack on Cryptographic Message Syntax", RFC 3218, January 2002.
- [RFC5652] Housley, R., "Cryptographic Message Syntax (CMS)", STD 70,

RFC 5652, September 2009.

Appendix A. JWE Examples

This section provides examples of JWE computations.

A.1. Example JWE using RSAES OAEP and AES GCM

This example encrypts the plaintext "The true sign of intelligence is not knowledge but imagination." to the recipient using RSAES OAEP for key encryption and AES GCM for content encryption. The representation of this plaintext is:

```
[84, 104, 101, 32, 116, 114, 117, 101, 32, 115, 105, 103, 110, 32,
111, 102, 32, 105, 110, 116, 101, 108, 108, 105, 103, 101, 110, 99,
101, 32, 105, 115, 32, 110, 111, 116, 32, 107, 110, 111, 119, 108,
101, 100, 103, 101, 32, 98, 117, 116, 32, 105, 109, 97, 103, 105,
110, 97, 116, 105, 111, 110, 46]
```

A.1.1. JWE Header

The following example JWE Protected Header declares that:

- o the Content Encryption Key is encrypted to the recipient using the RSAES OAEP algorithm to produce the JWE Encrypted Key and
- o the Plaintext is encrypted using the AES GCM algorithm with a 256 bit key to produce the Ciphertext.

```
{"alg":"RSA-OAEP","enc":"A256GCM"}
```

Encoding this JWE Protected Header as BASE64URL(UTF8(JWE Protected Header)) gives this value:

```
eyJhbGciOiJSU0EtT0FFUCIsImVuYyI6IkJEYNTZHQ00ifQ
```

A.1.2. Content Encryption Key (CEK)

Generate a 256 bit random Content Encryption Key (CEK). In this example, the value is:

```
[177, 161, 244, 128, 84, 143, 225, 115, 63, 180, 3, 255, 107, 154,
212, 246, 138, 7, 110, 91, 112, 46, 34, 105, 47, 130, 203, 46, 122,
234, 64, 252]
```


A.1.3. Key Encryption

Encrypt the CEK with the recipient's public key using the RSAES OAEP algorithm to produce the JWE Encrypted Key. This example uses the RSA key represented in JSON Web Key [JWK] format below (with line breaks for display purposes only):

```
{ "kty": "RSA",
  "n": "oahUIoWw0K0usKNuOR6H4wkf4oBUXHTxRvgb48E-BVvxkeDNjbC4he8rUW
cJoZmds2h7M70imEVhRU5djINXtqllXI4DFqcI1Dgjt9LewND8MW2Krf3S
psk_ZkoFnilakGygTwpZ3uesH-PFABNIUYpOiN15dsQRkgr0vEhxN92i2a
sbOenSZeyaxziK72UwxrrKoExv6kc5twXTq4h-QChLOln0_mtUZwfsRaMS
tPs6mS6XrgxnxbWhojf663tuEQueGC-FCMfra36C9knDFGzKsNa7LzK2dj
YgyD3JR_MB_4NUJW_TqOQtWYbxevoJArm-L5StowjzGy-_bq6Gw",
  "e": "AQAB",
  "d": "kLdtIj6GbDks_ApCSTYQtelcNttlKiOyPzMrXHeI-yk1F7-kpDxY4-WY5N
WV5KntaEeXS1j82E375xxhWMHXyvJecPT9fpwR_M9gV8n9Hrh2anTpTD9
3Dt62ypW3yDsJzBnTnrYuliwWRgBKREYY46qAZIrA2xAwnm2X7uGR1hghk
qDp0Vqj3kbSCz1XyfCs6_LehBwtxHIyh8Ripy40p24moOAbgxVw3rxT_vl
t3UVE4W03JkJOzlpUf-KTVI2Ptgm-dARxTETe-id-40Jr0h-K-VFs3VSnd
VTIznSxfyrj8ILL6MG_Uv8YAu7VILSB3lOW085-4qE3DzgrTjgyQ"
}
```

The resulting JWE Encrypted Key value is:

```
[56, 163, 154, 192, 58, 53, 222, 4, 105, 218, 136, 218, 29, 94, 203,
22, 150, 92, 129, 94, 211, 232, 53, 89, 41, 60, 138, 56, 196, 216,
82, 98, 168, 76, 37, 73, 70, 7, 36, 8, 191, 100, 136, 196, 244, 220,
145, 158, 138, 155, 4, 117, 141, 230, 199, 247, 173, 45, 182, 214,
74, 177, 107, 211, 153, 11, 205, 196, 171, 226, 162, 128, 171, 182,
13, 237, 239, 99, 193, 4, 91, 219, 121, 223, 107, 167, 61, 119, 228,
173, 156, 137, 134, 200, 80, 219, 74, 253, 56, 185, 91, 177, 34, 158,
89, 154, 205, 96, 55, 18, 138, 43, 96, 218, 215, 128, 124, 75, 138,
243, 85, 25, 109, 117, 140, 26, 155, 249, 67, 167, 149, 231, 100, 6,
41, 65, 214, 251, 232, 87, 72, 40, 182, 149, 154, 168, 31, 193, 126,
215, 89, 28, 111, 219, 125, 182, 139, 235, 195, 197, 23, 234, 55, 58,
63, 180, 68, 202, 206, 149, 75, 205, 248, 176, 67, 39, 178, 60, 98,
193, 32, 238, 122, 96, 158, 222, 57, 183, 111, 210, 55, 188, 215,
206, 180, 166, 150, 166, 106, 250, 55, 229, 72, 40, 69, 214, 216,
104, 23, 40, 135, 212, 28, 127, 41, 80, 175, 174, 168, 115, 171, 197,
89, 116, 92, 103, 246, 83, 216, 182, 176, 84, 37, 147, 35, 45, 219,
172, 99, 226, 233, 73, 37, 124, 42, 72, 49, 242, 35, 127, 184, 134,
117, 114, 135, 206]
```

Encoding this JWE Encrypted Key as BASE64URL(JWE Encrypted Key) gives this value (with line breaks for display purposes only):

```
OKOawDo13gRp2ojaHV7LFpZcgV7T6DVZKTyKOMTYUmKoTCVJRgckCL9kiMT03JGe  
ipsEdY3mx_etLbbWSrFr05kLzcSr4qKAq7YN7e9jwQRb23nfa6c9d-StnImGyFDb  
Sv04uVuxIp5ZmslgNxKKK2Da14B8S4rzVRltdYwam_lDp5XnZAYpQdb76FdIKLaV  
mqgfwX7XWRxv2322i-vDxRfqNzo_tETKzpVLzfiwQyeyPGLBIO56YJ7eObdv0je8  
1860ppamavo35UgoRdbYaBcoh9QcfylQr66oc6vFWXRcZ_ZT2LawVCWTIy3brGPi  
6UklfCpIMfIjf7iGdXKHZg
```

A.1.4. Initialization Vector

Generate a random 96 bit JWE Initialization Vector. In this example, the value is:

```
[227, 197, 117, 252, 2, 219, 233, 68, 180, 225, 77, 219]
```

Encoding this JWE Initialization Vector as BASE64URL(JWE Initialization Vector) gives this value:

```
48Vl_ALb6US04U3b
```

A.1.5. Additional Authenticated Data

Let the Additional Authenticated Data encryption parameter be ASCII(BASE64URL(UTF8(JWE Protected Header))). This value is:

```
[101, 121, 74, 104, 98, 71, 99, 105, 79, 105, 74, 83, 85, 48, 69,  
116, 84, 48, 70, 70, 85, 67, 73, 115, 73, 109, 86, 117, 89, 121, 73,  
54, 73, 107, 69, 121, 78, 84, 90, 72, 81, 48, 48, 105, 102, 81]
```

A.1.6. Content Encryption

Encrypt the Plaintext with AES GCM using the CEK as the encryption key, the JWE Initialization Vector, and the Additional Authenticated Data value above, requesting a 128 bit Authentication Tag output. The resulting Ciphertext is:

```
[229, 236, 166, 241, 53, 191, 115, 196, 174, 43, 73, 109, 39, 122,  
233, 96, 140, 206, 120, 52, 51, 237, 48, 11, 190, 219, 186, 80, 111,  
104, 50, 142, 47, 167, 59, 61, 181, 127, 196, 21, 40, 82, 242, 32,  
123, 143, 168, 226, 73, 216, 176, 144, 138, 247, 106, 60, 16, 205,  
160, 109, 64, 63, 192]
```

The resulting Authentication Tag value is:

```
[92, 80, 104, 49, 133, 25, 161, 215, 173, 101, 219, 211, 136, 91,  
210, 145]
```

Encoding this JWE Ciphertext as BASE64URL(JWE Ciphertext) gives this value (with line breaks for display purposes only):

```
5eym8TW_c8SuK0ltJ3rpYIzOeDQz7TALvtu6UG9oMo4vpzs9tX_EFShS8iB7j6ji
SdiwkIr3ajwQzaBtQD_A
```

Encoding this JWE Authentication Tag as BASE64URL(JWE Authentication Tag) gives this value:

```
XFBomYUZodetZdvTiFvSkQ
```

A.1.7. Complete Representation

Assemble the final representation: The Compact Serialization of this result is the string BASE64URL(UTF8(JWE Protected Header)) || '.' || BASE64URL(JWE Encrypted Key) || '.' || BASE64URL(JWE Initialization Vector) || '.' || BASE64URL(JWE Ciphertext) || '.' || BASE64URL(JWE Authentication Tag).

The final result in this example (with line breaks for display purposes only) is:

```
eyJhbGciOiJSU0EtT0FFUCIsImVuYyI6IkEyNTZHQ00ifQ.
OKOawDo13gRp2ojaHV7LFpZcgV7T6DVZKTyKOMTYUmKoTCVJRgckCL9kiMT03JGe
ipsEdY3mx_etLbbWSrFr05kLzcSr4qKAq7YN7e9jwQRb23nfa6c9d-StnImGyFDb
Sv04uVuxIp5ZmslgNxKKK2Da14B8S4rzVRltdYwam_lDp5XnZAYpQdb76FdIKLaV
mqgfwX7XWRxv2322i-vDxRfqNzo_tETKzpVLzfiwQyeyPGLBIO56YJ7eObdv0je8
1860ppamavo35UgoRdbYaBcoh9QcfylQr66oc6vFWXRcZ_ZT2LawVCWTIy3brGPi
6UklfCpIMfIjf7iGdXKHZg.
48Vl_ALb6US04U3b.
5eym8TW_c8SuK0ltJ3rpYIzOeDQz7TALvtu6UG9oMo4vpzs9tX_EFShS8iB7j6ji
SdiwkIr3ajwQzaBtQD_A.
XFBomYUZodetZdvTiFvSkQ
```

A.1.8. Validation

This example illustrates the process of creating a JWE with RSAES OAEP for key encryption and AES GCM for content encryption. These results can be used to validate JWE decryption implementations for these algorithms. Note that since the RSAES OAEP computation includes random values, the encryption results above will not be completely reproducible. However, since the AES GCM computation is deterministic, the JWE Encrypted Ciphertext values will be the same for all encryptions performed using these inputs.

A.2. Example JWE using RSAES-PKCS1-V1_5 and AES_128_CBC_HMAC_SHA_256

This example encrypts the plaintext "Live long and prosper." to the recipient using RSAES-PKCS1-V1_5 for key encryption and AES_128_CBC_HMAC_SHA_256 for content encryption. The representation of this plaintext is:

```
[76, 105, 118, 101, 32, 108, 111, 110, 103, 32, 97, 110, 100, 32,
112, 114, 111, 115, 112, 101, 114, 46]
```

A.2.1. JWE Header

The following example JWE Protected Header declares that:

- o the Content Encryption Key is encrypted to the recipient using the RSAES-PKCS1-V1_5 algorithm to produce the JWE Encrypted Key and
- o the Plaintext is encrypted using the AES_128_CBC_HMAC_SHA_256 algorithm to produce the Ciphertext.

```
{"alg":"RSA1_5","enc":"A128CBC-HS256"}
```

Encoding this JWE Protected Header as `BASE64URL(UTF8(JWE Protected Header))` gives this value:

```
eyJhbGciOiJSU0ExXzUiLCJlbmMiOiJBMTI4Q0JDLUhTMjU2In0
```

A.2.2. Content Encryption Key (CEK)

Generate a 256 bit random Content Encryption Key (CEK). In this example, the key value is:

```
[4, 211, 31, 197, 84, 157, 252, 254, 11, 100, 157, 250, 63, 170, 106,
206, 107, 124, 212, 45, 111, 107, 9, 219, 200, 177, 0, 240, 143, 156,
44, 207]
```

A.2.3. Key Encryption

Encrypt the CEK with the recipient's public key using the RSAES-PKCS1-V1_5 algorithm to produce the JWE Encrypted Key. This example uses the RSA key represented in JSON Web Key [JWK] format below (with line breaks for display purposes only):

```
{ "kty": "RSA",
  "n": "sXchDaQebHnPiGvyDOAT4saGEUetSyo9MKLOoWFsueri23bOdGwP4Dy1Wl
    UzewbgBHod5pcM9H95GQRV3JDXboIRROSBigeC5yJUlHGzHHyXss8UDpre
    cbAYxknTcQkhsLANGRUZmdTOQ5qTRsLat6BTYuyvVRdhS8exSZEy_c4gs_
    7svlJJQ4H9_NxsiIoLwAEk7-Q3UXERGYw_75IDrGA84-lA_-Ct4eTlXHBI
    Y2EaV7t7LjJaynVJCpkv4LKjTTAumiGUIuQhrNhZLuF_RJLqHpM2kgWFLU
    7-VTdLlVbC2tejvcI2BlMkEpk1BzBZI0KQB0GaDWFLN-aEAw3vRw",
  "e": "AQAB",
  "d": "VFCWOqXr8nvZNYaaJLXdnNPXZKRaWCjkU5Q2egQQpTBMwhprMzWzpR8Sxq
    lOPThh_J6MUD8Z35wky9b8eEO0pwNS8xlh1lOFRRBoNqDIKVoku0aZb-ry
    nq8cxjDTLZQ6Fz7jSjRlKlop-YKAUhc9GsEofQqYruPhzSA-QgajZGPbE_
    0ZaVDJHfyd7UUBUKunFMScbflYAAOYJqVIVwaYR5zWEEceUjNnTNo_CVSj
    -VvXLO5VZfCUAVLgW4dpf1SrtZjSt34YLSRarSb127reG_DUwg9Ch-KyvJ
    TlSkHgUWRVGcyly7uvVGRSDwsXypdrNinPA4jlhoNdizK2zf2CWQ"
}
```

The resulting JWE Encrypted Key value is:

```
[80, 104, 72, 58, 11, 130, 236, 139, 132, 189, 255, 205, 61, 86, 151,
176, 99, 40, 44, 233, 176, 189, 205, 70, 202, 169, 72, 40, 226, 181,
156, 223, 120, 156, 115, 232, 150, 209, 145, 133, 104, 112, 237, 156,
116, 250, 65, 102, 212, 210, 103, 240, 177, 61, 93, 40, 71, 231, 223,
226, 240, 157, 15, 31, 150, 89, 200, 215, 198, 203, 108, 70, 117, 66,
212, 238, 193, 205, 23, 161, 169, 218, 243, 203, 128, 214, 127, 253,
215, 139, 43, 17, 135, 103, 179, 220, 28, 2, 212, 206, 131, 158, 128,
66, 62, 240, 78, 186, 141, 125, 132, 227, 60, 137, 43, 31, 152, 199,
54, 72, 34, 212, 115, 11, 152, 101, 70, 42, 219, 233, 142, 66, 151,
250, 126, 146, 141, 216, 190, 73, 50, 177, 146, 5, 52, 247, 28, 197,
21, 59, 170, 247, 181, 89, 131, 241, 169, 182, 246, 99, 15, 36, 102,
166, 182, 172, 197, 136, 230, 120, 60, 58, 219, 243, 149, 94, 222,
150, 154, 194, 110, 227, 225, 112, 39, 89, 233, 112, 207, 211, 241,
124, 174, 69, 221, 179, 107, 196, 225, 127, 167, 112, 226, 12, 242,
16, 24, 28, 120, 182, 244, 213, 244, 153, 194, 162, 69, 160, 244,
248, 63, 165, 141, 4, 207, 249, 193, 79, 131, 0, 169, 233, 127, 167,
101, 151, 125, 56, 112, 111, 248, 29, 232, 90, 29, 147, 110, 169,
146, 114, 165, 204, 71, 136, 41, 252]
```

Encoding this JWE Encrypted Key as BASE64URL(JWE Encrypted Key) gives this value (with line breaks for display purposes only):

```
UGhIOguC7IuEvf_NPVaXsGMoLOmwvc1GyqlIKOKlnN94nHPoltGRhWhw7Zx0-kFm
1NJn8LE9XShH59_i8J0PH5ZZyNfGy2xGdULU7sHNF6Gp2vPLgNZ__deLKxGHZ7Pc
HALUzoOegEI-8E66jX2E4zyJKx-YxzZIIItRzC5hlRirb6Y5Cl_p-ko3YvkkysZIF
NPccxRU7qvelWYPxqbb2Yw8kZqa2rMWI5ng8OtvzlV7elprCbuPhcCdZ6XDP0_F8
rkXds2vE4X-ncOIM8hAYHHi29NX0mckIRaD0-D-ljQTP-cFPgwCp6X-nZZd9OHBv
-B3oWh2TbqmScqXMR4gp_A
```

A.2.4. Initialization Vector

Generate a random 128 bit JWE Initialization Vector. In this example, the value is:

```
[3, 22, 60, 12, 43, 67, 104, 105, 108, 108, 105, 99, 111, 116, 104, 101]
```

Encoding this JWE Initialization Vector as `BASE64URL(JWE Initialization Vector)` gives this value:

```
AxY8DCtDaGlsbGljb3RoZQ
```

A.2.5. Additional Authenticated Data

Let the Additional Authenticated Data encryption parameter be `ASCII(BASE64URL(UTF8(JWE Protected Header)))`. This value is:

```
[101, 121, 74, 104, 98, 71, 99, 105, 79, 105, 74, 83, 85, 48, 69, 120, 88, 122, 85, 105, 76, 67, 74, 108, 98, 109, 77, 105, 79, 105, 74, 66, 77, 84, 73, 52, 81, 48, 74, 68, 76, 85, 104, 84, 77, 106, 85, 50, 73, 110, 48]
```

A.2.6. Content Encryption

Encrypt the Plaintext with `AES_128_CBC_HMAC_SHA_256` using the CEK as the encryption key, the JWE Initialization Vector, and the Additional Authenticated Data value above. The steps for doing this using the values from Appendix A.3 are detailed in Appendix B. The resulting Ciphertext is:

```
[40, 57, 83, 181, 119, 33, 133, 148, 198, 185, 243, 24, 152, 230, 6, 75, 129, 223, 127, 19, 210, 82, 183, 230, 168, 33, 215, 104, 143, 112, 56, 102]
```

The resulting Authentication Tag value is:

```
[246, 17, 244, 190, 4, 95, 98, 3, 231, 0, 115, 157, 242, 203, 100, 191]
```

Encoding this JWE Ciphertext as `BASE64URL(JWE Ciphertext)` gives this value:

```
KDlTtXchhZTGufMYmOYGS4HffxPSUrfmqCHXaI9wOGY
```

Encoding this JWE Authentication Tag as `BASE64URL(JWE Authentication Tag)` gives this value:

9hH0vgRfYgPnAHOd8stkvw

A.2.7. Complete Representation

Assemble the final representation: The Compact Serialization of this result is the string `BASE64URL(UTF8(JWE Protected Header)) || '.' || BASE64URL(JWE Encrypted Key) || '.' || BASE64URL(JWE Initialization Vector) || '.' || BASE64URL(JWE Ciphertext) || '.' || BASE64URL(JWE Authentication Tag)`.

The final result in this example (with line breaks for display purposes only) is:

```
eyJhbGciOiJSU0ExXzUiLCJlbmMiOiJBMTI4Q0JDLUhTMjU2In0.
UGhIOguC7IuEvf_NPVaXsGMoLOmwvclGyqlIKOKlnN94nHPoltGRhWhw7Zx0-kFm
1Njn8LE9XShH59_i8J0PH5ZZyNfGy2xGdULU7shNF6Gp2vPLgNZ__deLKxGHZ7Pc
HALUzoOegEI-8E66jX2E4zyJKx-YxzZIItrZC5hlRirb6Y5Cl_p-ko3YvkkysZIF
NPccxRU7qve1WYPxqbb2Yw8kZqa2rMWI5ng8OtvzlV7elprCbuPhcCdZ6XDP0_F8
rkXds2vE4X-ncOIM8hAYHHI29NX0mcKiRaD0-D-1jQTP-cFPgwCp6X-nZZd9OHBv
-B3oWh2TbqmScqXMR4gp_A.
AxY8DCtDaGlsbGljb3RoZQ.
Kd1TtXchhZTGufMYmOYGS4HffxPSUrfmqCHXaI9wOGY.
9hH0vgRfYgPnAHOd8stkvw
```

A.2.8. Validation

This example illustrates the process of creating a JWE with `RSAES-PKCS1-V1_5` for key encryption and `AES_CBC_HMAC_SHA2` for content encryption. These results can be used to validate JWE decryption implementations for these algorithms. Note that since the `RSAES-PKCS1-V1_5` computation includes random values, the encryption results above will not be completely reproducible. However, since the `AES CBC` computation is deterministic, the `JWE Encrypted Ciphertext` values will be the same for all encryptions performed using these inputs.

A.3. Example JWE using AES Key Wrap and AES_128_CBC_HMAC_SHA_256

This example encrypts the plaintext "Live long and prosper." to the recipient using `AES Key Wrap` for key encryption and `AES GCM` for content encryption. The representation of this plaintext is:

```
[76, 105, 118, 101, 32, 108, 111, 110, 103, 32, 97, 110, 100, 32,
112, 114, 111, 115, 112, 101, 114, 46]
```

A.3.1. JWE Header

The following example `JWE Protected Header` declares that:

- o the Content Encryption Key is encrypted to the recipient using the AES Key Wrap algorithm with a 128 bit key to produce the JWE Encrypted Key and
- o the Plaintext is encrypted using the AES_128_CBC_HMAC_SHA_256 algorithm to produce the Ciphertext.

```
{"alg": "A128KW", "enc": "A128CBC-HS256"}
```

Encoding this JWE Protected Header as BASE64URL(UTF8(JWE Protected Header)) gives this value:

```
eyJhbGciOiJBMTI4S1ciLCJlbmMiOiJBMTI4Q0JDLUhTMjU2In0
```

A.3.2. Content Encryption Key (CEK)

Generate a 256 bit random Content Encryption Key (CEK). In this example, the value is:

```
[4, 211, 31, 197, 84, 157, 252, 254, 11, 100, 157, 250, 63, 170, 106,
206, 107, 124, 212, 45, 111, 107, 9, 219, 200, 177, 0, 240, 143, 156,
44, 207]
```

A.3.3. Key Encryption

Encrypt the CEK with the shared symmetric key using the AES Key Wrap algorithm to produce the JWE Encrypted Key. This example uses the symmetric key represented in JSON Web Key [JWK] format below:

```
{"kty": "oct",
 "k": "GawggufyGrWKav7AX4VKUg"
}
```

The resulting JWE Encrypted Key value is:

```
[232, 160, 123, 211, 183, 76, 245, 132, 200, 128, 123, 75, 190, 216,
22, 67, 201, 138, 193, 186, 9, 91, 122, 31, 246, 90, 28, 139, 57, 3,
76, 124, 193, 11, 98, 37, 173, 61, 104, 57]
```

Encoding this JWE Encrypted Key as BASE64URL(JWE Encrypted Key) gives this value:

```
6KB707dM9YTIgHtLvtgWQ8mKwboJW3of9locizkDTHzBC2IlrTlOQ
```


A.3.4. Initialization Vector

Generate a random 128 bit JWE Initialization Vector. In this example, the value is:

```
[3, 22, 60, 12, 43, 67, 104, 105, 108, 108, 105, 99, 111, 116, 104, 101]
```

Encoding this JWE Initialization Vector as `BASE64URL(JWE Initialization Vector)` gives this value:

```
AxY8DCtDaGlsbGljb3RoZQ
```

A.3.5. Additional Authenticated Data

Let the Additional Authenticated Data encryption parameter be `ASCII(BASE64URL(UTF8(JWE Protected Header)))`. This value is:

```
[101, 121, 74, 104, 98, 71, 99, 105, 79, 105, 74, 66, 77, 84, 73, 52, 83, 49, 99, 105, 76, 67, 74, 108, 98, 109, 77, 105, 79, 105, 74, 66, 77, 84, 73, 52, 81, 48, 74, 68, 76, 85, 104, 84, 77, 106, 85, 50, 73, 110, 48]
```

A.3.6. Content Encryption

Encrypt the Plaintext with `AES_128_CBC_HMAC_SHA_256` using the CEK as the encryption key, the JWE Initialization Vector, and the Additional Authenticated Data value above. The steps for doing this using the values from this example are detailed in Appendix B. The resulting Ciphertext is:

```
[40, 57, 83, 181, 119, 33, 133, 148, 198, 185, 243, 24, 152, 230, 6, 75, 129, 223, 127, 19, 210, 82, 183, 230, 168, 33, 215, 104, 143, 112, 56, 102]
```

The resulting Authentication Tag value is:

```
[83, 73, 191, 98, 104, 205, 211, 128, 201, 189, 199, 133, 32, 38, 194, 85]
```

Encoding this JWE Ciphertext as `BASE64URL(JWE Ciphertext)` gives this value:

```
KDlTtXchhZTGufMYmOYGS4HffxPSUrfmqCHXaI9wOGY
```

Encoding this JWE Authentication Tag as `BASE64URL(JWE Authentication Tag)` gives this value:

U0m_YmjN04DJvceFICbCVQ

A.3.7. Complete Representation

Assemble the final representation: The Compact Serialization of this result is the string `BASE64URL(UTF8(JWE Protected Header)) || '.' || BASE64URL(JWE Encrypted Key) || '.' || BASE64URL(JWE Initialization Vector) || '.' || BASE64URL(JWE Ciphertext) || '.' || BASE64URL(JWE Authentication Tag)`.

The final result in this example (with line breaks for display purposes only) is:

```
eyJhbGciOiJBMTI4S1ciLCJlbmMiOiJBMTI4Q0JDLUhTMjU2In0.
6KB707dM9YTIgHtLvtgWQ8mKwboJW3of9locizkDTHzBC2IlrT1oOQ.
AxY8DCtDaGlsbGljb3RoZQ.
KDlTtXchhZTGufMYmOYGS4HffxPSUrfmqCHXaI9wOGY.
U0m_YmjN04DJvceFICbCVQ
```

A.3.8. Validation

This example illustrates the process of creating a JWE with AES Key Wrap for key encryption and AES GCM for content encryption. These results can be used to validate JWE decryption implementations for these algorithms. Also, since both the AES Key Wrap and AES GCM computations are deterministic, the resulting JWE value will be the same for all encryptions performed using these inputs. Since the computation is reproducible, these results can also be used to validate JWE encryption implementations for these algorithms.

A.4. Example JWE using JWE JSON Serialization

This section contains an example using the JWE JSON Serialization. This example demonstrates the capability for encrypting the same plaintext to multiple recipients.

Two recipients are present in this example. The algorithm and key used for the first recipient are the same as that used in Appendix A.2. The algorithm and key used for the second recipient are the same as that used in Appendix A.3. The resulting JWE Encrypted Key values are therefore the same; those computations are not repeated here.

The Plaintext, the Content Encryption Key (CEK), Initialization Vector, and JWE Protected Header are shared by all recipients (which must be the case, since the Ciphertext and Authentication Tag are also shared).

A.4.1. JWE Per-Recipient Unprotected Headers

The first recipient uses the RSAES-PKCS1-V1_5 algorithm to encrypt the Content Encryption Key (CEK). The second uses AES Key Wrap to encrypt the CEK. Key ID values are supplied for both keys. The two per-recipient header values used to represent these algorithms and Key IDs are:

```
{"alg": "RSA1_5", "kid": "2011-04-29"}
```

and

```
{"alg": "A128KW", "kid": "7"}
```

A.4.2. JWE Protected Header

The Plaintext is encrypted using the AES_128_CBC_HMAC_SHA_256 algorithm to produce the common JWE Ciphertext and JWE Authentication Tag values. The JWE Protected Header value representing this is:

```
{"enc": "A128CBC-HS256"}
```

Encoding this JWE Protected Header as `BASE64URL(UTF8(JWE Protected Header))` gives this value:

```
eyJlbmMiOiJBMTI4Q0JDLUhTMjU2In0
```

A.4.3. JWE Unprotected Header

This JWE uses the "jku" Header Parameter to reference a JWK Set. This is represented in the following JWE Unprotected Header value as:

```
{"jku": "https://server.example.com/keys.jwks"}
```

A.4.4. Complete JWE Header Values

Combining the per-recipient, protected, and unprotected header values supplied, the JWE Header values used for the first and second recipient respectively are:

```
{ "alg": "RSA1_5",  
  "kid": "2011-04-29",  
  "enc": "A128CBC-HS256",  
  "jku": "https://server.example.com/keys.jwks" }
```

and

```
{ "alg": "A128KW",  
  "kid": "7",  
  "enc": "A128CBC-HS256",  
  "jku": "https://server.example.com/keys.jwks" }
```

A.4.5. Additional Authenticated Data

Let the Additional Authenticated Data encryption parameter be ASCII(BASE64URL(UTF8(JWE Protected Header))). This value is:

```
[101, 121, 74, 108, 98, 109, 77, 105, 79, 105, 74, 66, 77, 84, 73,  
52, 81, 48, 74, 68, 76, 85, 104, 84, 77, 106, 85, 50, 73, 110, 48]
```

A.4.6. Content Encryption

Encrypt the Plaintext with AES_128_CBC_HMAC_SHA_256 using the CEK as the encryption key, the JWE Initialization Vector, and the Additional Authenticated Data value above. The steps for doing this using the values from Appendix A.3 are detailed in Appendix B. The resulting Ciphertext is:

```
[40, 57, 83, 181, 119, 33, 133, 148, 198, 185, 243, 24, 152, 230, 6,  
75, 129, 223, 127, 19, 210, 82, 183, 230, 168, 33, 215, 104, 143,  
112, 56, 102]
```

The resulting Authentication Tag value is:

```
[51, 63, 149, 60, 252, 148, 225, 25, 92, 185, 139, 245, 35, 2, 47,  
207]
```

Encoding this JWE Ciphertext as BASE64URL(JWE Ciphertext) gives this value:

```
KDlTtXchhZTGufMYmOYGS4HffxPSUrfmqCHXaI9wOGY
```

Encoding this JWE Authentication Tag as BASE64URL(JWE Authentication Tag) gives this value:

```
Mz-VPPyU4RlcuYv1IwIvzw
```

A.4.7. Complete JWE JSON Serialization Representation

The complete JSON Web Encryption JSON Serialization for these values is as follows (with line breaks for display purposes only):

```

{
  "protected":
    "eyJlbmMiOiJBMTI4Q0JDLUhTMjU2In0",
  "unprotected":
    {
      "jku": "https://server.example.com/keys.jwks",
      "recipients": [
        {
          "header":
            {
              "alg": "RSA1_5",
              "encrypted_key":
                "UGhIOguC7IuEvf_NPVaXsGMOLOmwvc1GyqlIKOKlnN94nHPoltGRhWhw7Zx0-
                kFm1NJn8LE9XShH59_i8J0PH5ZZyNfGy2xGdULU7sHNF6Gp2vPLgNZ__deLKx
                GHZ7PcHALUzoOegEI-8E66jX2E4zyJKx-YxzZIItrZC5hlRirb6Y5Cl_p-ko3
                YvkkysZIFNPccxRU7qvelWYPxqbb2Yw8kZqa2rMWI5ng8OtvzlV7elprCbuPh
                cCdZ6XDP0_F8rkXds2vE4X-ncOIM8hAYHHi29NX0mcKiRaD0-D-1jQTP-cFPg
                wCp6X-nZZd9OHBv-B3oWh2TbqmScqXMR4gp_A",
            },
          "header":
            {
              "alg": "A128KW",
              "encrypted_key":
                "6KB707dM9YTIgHtLvtgWQ8mKwboJW3of9locizkDTHzBC2IlrT1oOQ"
            }
        },
        {
          "iv":
            "Axy8DCtDaGlsbGljb3RoZQ",
          "ciphertext":
            "Kd1TtXchhZTGufMYmOYGS4HffxPSUrfmqCHXaI9wOGY",
          "tag":
            "Mz-VPPyU4RlcuYv1IwIvzw"
        }
      ]
    }
  }

```

Appendix B. Example AES_128_CBC_HMAC_SHA_256 Computation

This example shows the steps in the AES_128_CBC_HMAC_SHA_256 authenticated encryption computation using the values from the example in Appendix A.3. As described where this algorithm is defined in Sections 4.8 and 4.8.3 of JWA, the AES_CBC_HMAC_SHA2 family of algorithms are implemented using Advanced Encryption Standard (AES) in Cipher Block Chaining (CBC) mode with PKCS #5 padding to perform the encryption and an HMAC SHA-2 function to perform the integrity calculation - in this case, HMAC SHA-256.

B.1. Extract MAC_KEY and ENC_KEY from Key

The 256 bit AES_128_CBC_HMAC_SHA_256 key K used in this example is:

```
[4, 211, 31, 197, 84, 157, 252, 254, 11, 100, 157, 250, 63, 170, 106,
206, 107, 124, 212, 45, 111, 107, 9, 219, 200, 177, 0, 240, 143, 156,
44, 207]
```

Use the first 128 bits of this key as the HMAC SHA-256 key MAC_KEY, which is:

```
[4, 211, 31, 197, 84, 157, 252, 254, 11, 100, 157, 250, 63, 170, 106, 206]
```

Use the last 128 bits of this key as the AES CBC key ENC_KEY, which is:

```
[107, 124, 212, 45, 111, 107, 9, 219, 200, 177, 0, 240, 143, 156, 44, 207]
```

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 identifiers "AES_128_CBC_HMAC_SHA_256" and "A128CBC-HS256".

B.2. Encrypt Plaintext to Create Ciphertext

Encrypt the Plaintext with AES in Cipher Block Chaining (CBC) mode using PKCS #5 padding using the ENC_KEY above. The Plaintext in this example is:

```
[76, 105, 118, 101, 32, 108, 111, 110, 103, 32, 97, 110, 100, 32, 112, 114, 111, 115, 112, 101, 114, 46]
```

The encryption result is as follows, which is the Ciphertext output:

```
[40, 57, 83, 181, 119, 33, 133, 148, 198, 185, 243, 24, 152, 230, 6, 75, 129, 223, 127, 19, 210, 82, 183, 230, 168, 33, 215, 104, 143, 112, 56, 102]
```

B.3. 64 Bit Big Endian Representation of AAD Length

The Additional Authenticated Data (AAD) in this example is:

```
[101, 121, 74, 104, 98, 71, 99, 105, 79, 105, 74, 66, 77, 84, 73, 52, 83, 49, 99, 105, 76, 67, 74, 108, 98, 109, 77, 105, 79, 105, 74, 66, 77, 84, 73, 52, 81, 48, 74, 68, 76, 85, 104, 84, 77, 106, 85, 50, 73, 110, 48]
```

This AAD is 51 bytes long, which is 408 bits long. The octet string AL, which is the number of bits in AAD expressed as a big endian 64 bit unsigned integer is:

```
[0, 0, 0, 0, 0, 0, 1, 152]
```

B.4. Initialization Vector Value

The Initialization Vector value used in this example is:

```
[3, 22, 60, 12, 43, 67, 104, 105, 108, 108, 105, 99, 111, 116, 104,
```

101]

B.5. Create Input to HMAC Computation

Concatenate the AAD, the Initialization Vector, the Ciphertext, and the AL value. The result of this concatenation is:

```
[101, 121, 74, 104, 98, 71, 99, 105, 79, 105, 74, 66, 77, 84, 73, 52,
83, 49, 99, 105, 76, 67, 74, 108, 98, 109, 77, 105, 79, 105, 74, 66,
77, 84, 73, 52, 81, 48, 74, 68, 76, 85, 104, 84, 77, 106, 85, 50, 73,
110, 48, 3, 22, 60, 12, 43, 67, 104, 105, 108, 108, 105, 99, 111,
116, 104, 101, 40, 57, 83, 181, 119, 33, 133, 148, 198, 185, 243, 24,
152, 230, 6, 75, 129, 223, 127, 19, 210, 82, 183, 230, 168, 33, 215,
104, 143, 112, 56, 102, 0, 0, 0, 0, 0, 0, 1, 152]
```

B.6. Compute HMAC Value

Compute the HMAC SHA-256 of the concatenated value above. This result M is:

```
[83, 73, 191, 98, 104, 205, 211, 128, 201, 189, 199, 133, 32, 38,
194, 85, 9, 84, 229, 201, 219, 135, 44, 252, 145, 102, 179, 140, 105,
86, 229, 116]
```

B.7. Truncate HMAC Value to Create Authentication Tag

Use the first half (128 bits) of the HMAC output M as the Authentication Tag output T. This truncated value is:

```
[83, 73, 191, 98, 104, 205, 211, 128, 201, 189, 199, 133, 32, 38,
194, 85]
```

Appendix C. Acknowledgements

Solutions for encrypting JSON content were also explored by JSON Simple Encryption [JSE] and JavaScript Message Security Format [I-D.rescorla-jsms], both of which significantly influenced this draft. This draft attempts to explicitly reuse as many of the relevant concepts from XML Encryption 1.1 [W3C.CR-xmlenc-core1-20120313] and RFC 5652 [RFC5652] as possible, while utilizing simple, compact JSON-based data structures.

Special thanks are due to John Bradley and Nat Sakimura for the discussions that helped inform the content of this specification and to Eric Rescorla and Joe Hildebrand for allowing the reuse of text from [I-D.rescorla-jsms] in this document.

Thanks to Axel Nennker, Emmanuel Raviart, Brian Campbell, and Edmund Jay for validating the examples in this specification.

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:

Richard Barnes, John Bradley, Brian Campbell, Breno de Medeiros, Dick Hardt, Jeff Hodges, Edmund Jay, James Manger, Matt Miller, Tony Nadalin, Axel Nennker, 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 D. Document History

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

-17

- o Refined the "typ" and "cty" definitions to always be MIME Media Types, with the omission of "application/" prefixes recommended for brevity, addressing issue #50.
- o Updated the mandatory-to-implement (MTI) language to say that general-purpose implementations must implement the single recipient case for both serializations whereas special-purpose implementations can implement just one serialization if that meets the needs of the use cases the implementation is designed for, addressing issue #176.
- o Explicitly named all the logical components of a JWE and defined the processing rules and serializations in terms of those components, addressing issues #60, #61, and #62.
- o Replaced verbose repetitive phrases such as "base64url encode the octets of the UTF-8 representation of X" with mathematical notation such as "BASE64URL(UTF8(X))".
- o Header Parameters and processing rules occurring in both JWS and JWE are now referenced in JWS by JWE, rather than duplicated, addressing issue #57.

- o 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.

-16

- o Changes to address editorial and minor issues #163, #168, #169, #170, #172, and #173.

-15

- o Clarified that it is an application decision which recipients' encrypted content must successfully validate for the JWE to be accepted, addressing issue #35.
- o Changes to address editorial issues #34, #164, and #169.

-14

- o Clarified that the "protected", "unprotected", "header", "iv", "tag", and "encrypted_key" parameters are to be omitted in the JWE JSON Serialization when their values would be empty. Stated that the "recipients" array must always be present.

-13

- o Added an "aad" (Additional Authenticated Data) member for the JWE JSON Serialization, enabling Additional Authenticated Data to be supplied that is not double base64url encoded, addressing issue #29.

-12

- o Clarified that the "typ" and "cty" header parameters are used in an application-specific manner and have no effect upon the JWE processing.
- o Replaced the MIME types "application/jwe+json" and "application/jwe" with "application/jose+json" and "application/jose".
- o Stated that recipients MUST either reject JWEs with duplicate Header Parameter Names or use a JSON parser that returns only the lexically last duplicate member name.
- o Moved the "epk", "apu", and "apv" Header Parameter definitions to be with the algorithm descriptions that use them.

- o Added a Serializations section with parallel treatment of the JWE Compact Serialization and the JWE JSON Serialization and also moved the former Implementation Considerations content there.
- o Restored use of the term "AEAD".
- o Changed terminology from "block encryption" to "content encryption".

-11

- o Added Key Identification section.
- o Removed the Encrypted Key value from the AAD computation since it is already effectively integrity protected by the encryption process. The AAD value now only contains the representation of the JWE Encrypted Header.
- o For the JWE JSON Serialization, enable Header Parameter values to be specified in any of three parameters: the "protected" member that is integrity protected and shared among all recipients, the "unprotected" member that is not integrity protected and shared among all recipients, and the "header" member that is not integrity protected and specific to a particular recipient. (This does not affect the JWE Compact Serialization, in which all Header Parameter values are in a single integrity protected JWE Header value.)
- o Shortened the names "authentication_tag" to "tag" and "initialization_vector" to "iv" in the JWE JSON Serialization, addressing issue #20.
- o Removed "apv" (agreement PartyVInfo) since it is no longer used.
- o Removed suggested compact serialization for multiple recipients.
- o Changed the MIME type name "application/jwe-js" to "application/jwe+json", addressing issue #22.
- o Tightened the description of the "crit" (critical) header parameter.

-10

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

- o Added an appendix suggesting a possible compact serialization for JWEs with multiple recipients.

-09

- o Added JWE JSON Serialization, as specified by draft-jones-jose-jwe-json-serialization-04.
- o Registered "application/jwe-js" MIME type and "JWE-JS" typ header parameter value.
- o Defined that the default action for header parameters that are not understood is to ignore them unless specifically designated as "MUST be understood" or included in the new "crit" (critical) header parameter list. This addressed issue #6.
- o Corrected "x5c" description. This addressed issue #12.
- o Changed from using the term "byte" to "octet" when referring to 8 bit values.
- o Added Key Management Mode definitions to terminology section and used the defined terms to provide clearer key management instructions. This addressed issue #5.
- o Added text about preventing the recipient from behaving as an oracle during decryption, especially when using RSAES-PKCS1-V1_5.
- o Changed from using the term "Integrity Value" to "Authentication Tag".
- o Changed member name from "integrity_value" to "authentication_tag" in the JWE JSON Serialization.
- o Removed Initialization Vector from the AAD value since it is already integrity protected by all of the authenticated encryption algorithms specified in the JWA specification.
- o Replaced "A128CBC+HS256" and "A256CBC+HS512" with "A128CBC-HS256" and "A256CBC-HS512". The new algorithms perform the same cryptographic computations as [I-D.mcgrewe-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.

-08

- o Replaced uses of the term "AEAD" with "Authenticated Encryption", since the term AEAD in the RFC 5116 sense implied the use of a particular data representation, rather than just referring to the class of algorithms that perform authenticated encryption with associated data.
- o Applied editorial improvements suggested by Jeff Hodges and Hannes Tschofenig. Many of these simplified the terminology used.
- o Clarified statements of the form "This header parameter is OPTIONAL" to "Use of this header parameter is OPTIONAL".
- o Added a Header Parameter Usage Location(s) field to the IANA JSON Web Signature and Encryption Header Parameters registry.
- o Added seriesInfo information to Internet Draft references.

-07

- o Added a data length prefix to PartyUInfo and PartyVInfo values.
- o Updated values for example AES CBC calculations.
- o Made several local editorial changes to clean up loose ends left over from the decision to only support block encryption methods providing integrity. One of these changes was to explicitly state that the "enc" (encryption method) algorithm must be an Authenticated Encryption algorithm with a specified key length.

-06

- o Removed the "int" and "kdf" parameters and defined the new composite Authenticated Encryption algorithms "A128CBC+HS256" and "A256CBC+HS512" to replace the former uses of AES CBC, which required the use of separate integrity and key derivation functions.
- o Included additional values in the Concat KDF calculation -- the desired output size and the algorithm value, and optionally PartyUInfo and PartyVInfo values. Added the optional header parameters "apu" (agreement PartyUInfo), "apv" (agreement PartyVInfo), "epu" (encryption PartyUInfo), and "epv" (encryption PartyVInfo). Updated the KDF examples accordingly.
- o Promoted Initialization Vector from being a header parameter to being a top-level JWE element. This saves approximately 16 bytes in the compact serialization, which is a significant savings for some use cases. Promoting the Initialization Vector out of the

header also avoids repeating this shared value in the JSON serialization.

- o Changed "x5c" (X.509 Certificate Chain) representation from being a single string to being an array of strings, each containing a single base64 encoded DER certificate value, representing elements of the certificate chain.
- o Added an AES Key Wrap example.
- o Reordered the encryption steps so CMK creation is first, when required.
- o Correct statements in examples about which algorithms produce reproducible results.

-05

- o Support both direct encryption using a shared or agreed upon symmetric key, and the use of a shared or agreed upon symmetric key to key wrap the CMK.
- o Added statement that "StringOrURI values are compared as case-sensitive strings with no transformations or canonicalizations applied".
- o Updated open issues.
- o Indented artwork elements to better distinguish them from the body text.

-04

- o Refer to the registries as the primary sources of defined values and then secondarily reference the sections defining the initial contents of the registries.
- o Normatively reference XML Encryption 1.1 [W3C.CR-xmlenc-core1-20120313] for its security considerations.
- o Reference draft-jones-jose-jwe-json-serialization instead of draft-jones-json-web-encryption-json-serialization.
- o Described additional open issues.
- o Applied editorial suggestions.

-03

- o Added the "kdf" (key derivation function) header parameter to provide crypto agility for key derivation. The default KDF remains the Concat KDF with the SHA-256 digest function.
- o Reordered encryption steps so that the Encoded JWE Header is always created before it is needed as an input to the Authenticated Encryption "additional authenticated data" parameter.
- o Added the "cty" (content type) header parameter for declaring type information about the secured content, as opposed to the "typ" (type) header parameter, which declares type information about this object.
- o Moved description of how to determine whether a header is for a JWS or a JWE from the JWT spec to the JWE spec.
- o Added complete encryption examples for both Authenticated Encryption and non-Authenticated Encryption algorithms.
- o Added complete key derivation examples.
- o Added "Collision Resistant Namespace" to the terminology section.
- o Reference ITU.X690.1994 for DER encoding.
- o Added Registry Contents sections to populate registry values.
- o Numerous editorial improvements.

-02

- o When using Authenticated Encryption algorithms (such as AES GCM), use the "additional authenticated data" parameter to provide integrity for the header, encrypted key, and ciphertext and use the resulting "authentication tag" value as the JWE Authentication Tag.
- o Defined KDF output key sizes.
- o Generalized text to allow key agreement to be employed as an alternative to key wrapping or key encryption.
- o Changed compression algorithm from gzip to DEFLATE.
- o Clarified that it is an error when a "kid" value is included and no matching key is found.

- o Clarified that JWEs with duplicate Header Parameter Names MUST be rejected.
- o Clarified the relationship between "typ" header parameter values and MIME types.
- o Registered application/jwe MIME type and "JWE" typ header parameter value.
- o Simplified JWK terminology to get replace the "JWK Key Object" and "JWK Container Object" terms with simply "JSON Web Key (JWK)" and "JSON Web Key Set (JWK Set)" and to eliminate potential confusion between single keys and sets of keys. As part of this change, the Header Parameter Name for a public key value was changed from "jpk" (JSON Public Key) to "jwk" (JSON Web Key).
- o Added suggestion on defining additional header parameters such as "x5t#S256" in the future for certificate thumbprints using hash algorithms other than SHA-1.
- o Specify RFC 2818 server identity validation, rather than RFC 6125 (paralleling the same decision in the OAuth specs).
- o Generalized language to refer to Message Authentication Codes (MACs) rather than Hash-based Message Authentication Codes (HMACs) unless in a context specific to HMAC algorithms.
- o Reformatted to give each header parameter its own section heading.

-01

- o Added an integrity check for non-Authenticated Encryption algorithms.
- o Added "jpk" and "x5c" header parameters for including JWK public keys and X.509 certificate chains directly in the header.
- o Clarified that this specification is defining the JWE Compact Serialization. Referenced the new JWE-JS spec, which defines the JWE JSON Serialization.
- o Added text "New header parameters should be introduced sparingly since an implementation that does not understand a parameter MUST reject the JWE".
- o Clarified that the order of the encryption and decryption steps is not significant in cases where there are no dependencies between the inputs and outputs of the steps.

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

-00

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

Authors' Addresses

Michael B. Jones
Microsoft

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

Eric Rescorla
RTFM, Inc.

Email: ekr@rtfm.com

Joe Hildebrand
Cisco Systems, Inc.

Email: jhildebr@cisco.com

JOSE Working Group
Internet-Draft
Intended status: Standards Track
Expires: April 10, 2014

M. Jones
Microsoft
October 7, 2013

JSON Web Key (JWK)
draft-ietf-jose-json-web-key-17

Abstract

A JSON Web Key (JWK) is a JavaScript Object Notation (JSON) data structure that represents a cryptographic key. This specification also defines a JSON Web Key Set (JWK Set) JSON data structure for representing a set of JWKs. Cryptographic algorithms and identifiers for use with this specification are described in the separate JSON Web Algorithms (JWA) specification and IANA registries defined by that specification.

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 April 10, 2014.

Copyright Notice

Copyright (c) 2013 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<http://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of

the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction	4
1.1. Notational Conventions	4
2. Terminology	4
3. JSON Web Key (JWK) Format	5
3.1. "kty" (Key Type) Parameter	5
3.2. "use" (Key Use) Parameter	6
3.3. "alg" (Algorithm) Parameter	6
3.4. "kid" (Key ID) Parameter	6
3.5. "x5u" (X.509 URL) Parameter	6
3.6. "x5t" (X.509 Certificate SHA-1 Thumbprint) Parameter	7
3.7. "x5c" (X.509 Certificate Chain) Parameter	7
4. JSON Web Key Set (JWK Set) Format	7
4.1. "keys" Parameter	8
5. String Comparison Rules	8
6. Encrypted JWK and Encrypted JWK Set Formats	8
7. IANA Considerations	9
7.1. JSON Web Key Parameters Registry	9
7.1.1. Registration Template	10
7.1.2. Initial Registry Contents	10
7.2. JSON Web Key Use Registry	11
7.2.1. Registration Template	12
7.2.2. Initial Registry Contents	12
7.3. JSON Web Key Set Parameters Registry	12
7.3.1. Registration Template	12
7.3.2. Initial Registry Contents	13
7.4. Media Type Registration	13
7.4.1. Registry Contents	13
8. Security Considerations	14
9. References	15
9.1. Normative References	15
9.2. Informative References	16
Appendix A. Example JSON Web Key Sets	17
A.1. Example Public Keys	17
A.2. Example Private Keys	17
A.3. Example Symmetric Keys	19
Appendix B. Example Use of "x5c" (X.509 Certificate Chain) Parameter	19
Appendix C. Example Encrypted RSA Private Key	20
C.1. Plaintext RSA Private Key	21
C.2. JWE Header	24
C.3. Content Encryption Key (CEK)	24
C.4. Key Encryption	25

C.5. Initialization Vector	25
C.6. Additional Authenticated Data	25
C.7. Content Encryption	26
C.8. Complete Representation	29
Appendix D. Acknowledgements	30
Appendix E. Document History	31
Author's Address	35

1. Introduction

A JSON Web Key (JWK) is a JavaScript Object Notation (JSON) [RFC4627] data structure that represents a cryptographic key. This specification also defines a JSON Web Key Set (JWK Set) JSON data structure for representing a set of JWKs. Cryptographic algorithms and identifiers for use with this specification are described in the separate JSON Web Algorithms (JWA) [JWA] specification and IANA registries defined by that specification.

Goals for this specification do not include representing certificate chains, representing certified keys, and replacing X.509 certificates.

JWKs and JWK Sets are used in the JSON Web Signature (JWS) [JWS] and JSON Web Encryption (JWE) [JWE] specifications.

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: "Base64url Encoding" and "Collision Resistant Name".

These terms are defined for use by this specification:

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

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

3. JSON Web Key (JWK) Format

A JSON Web Key (JWK) is a JSON object containing specific members, as specified below. Those members that are common to multiple key types are defined below.

In addition to the common parameters, each JWK will have members that are specific to the kind of key being represented. These members represent the parameters of the key. Section 5 of the JSON Web Algorithms (JWA) [JWA] specification defines multiple kinds of cryptographic keys and their associated members.

The member names within a JWK MUST be unique; recipients MUST either reject JWKs with duplicate member names or use a JSON parser that returns only the lexically last duplicate member name, as specified in Section 15.12 (The JSON Object) of ECMAScript 5.1 [ECMAScript].

Additional members MAY be present in the JWK. If not understood by implementations encountering them, they MUST be ignored. Member names used for representing key parameters for different kinds of keys need not be distinct. Any new member name SHOULD either be registered in the IANA JSON Web Key Parameters registry defined in Section 7.1 or be a value that contains a Collision Resistant Name.

3.1. "kty" (Key Type) Parameter

The "kty" (key type) member identifies the cryptographic algorithm family used with the key. "kty" values SHOULD either be registered in the IANA JSON Web Key Types registry defined in [JWA] or be a value that contains a Collision Resistant Name. The "kty" value is a case sensitive string. Use of this member is REQUIRED.

A list of defined "kty" values can be found in the IANA JSON Web Key Types registry defined in [JWA]; the initial contents of this registry are the values defined in Section 5.1 of the JSON Web Algorithms (JWA) [JWA] specification.

Additional members used with these "kty" values can be found in the IANA JSON Web Key Parameters registry defined in Section 7.1; the initial contents of this registry are the values defined in Sections 5.2 and 5.3 of the JSON Web Algorithms (JWA) [JWA] specification.

3.2. "use" (Key Use) Parameter

The "use" (key use) member identifies the intended use of the key. Values defined by this specification are:

- o "sig" (signature or MAC operation)
- o "enc" (encryption)

Other values MAY be used. Key Use values can be registered in the IANA JSON Web Key Use registry defined in Section 7.2. The "use" value is a case sensitive string. A "use" member SHOULD be present, unless the application uses another means or convention to determine the intended key usage.

When a key is used to wrap another key and a key use designation for the first key is desired, the "enc" (encryption) key use value SHOULD be used, since key wrapping is a kind of encryption. (The "alg" member can be used to specify the particular kind of encryption to be performed, when desired.)

3.3. "alg" (Algorithm) Parameter

The "alg" (algorithm) member identifies the algorithm intended for use with the key. The values used SHOULD either be registered in the IANA JSON Web Signature and Encryption Algorithms registry defined in [JWA] or be a value that contains a Collision Resistant Name. Use of this member is OPTIONAL.

3.4. "kid" (Key ID) Parameter

The "kid" (key ID) member can be used to match a specific key. This can be used, for instance, to choose among a set of keys within a JWK Set during key rollover. The interpretation of the "kid" value is unspecified. When "kid" values are used within a JWK Set, different keys within the JWK Set SHOULD use distinct "kid" values. The "kid" value is a case sensitive string. Use of this member is OPTIONAL.

When used with JWS or JWE, the "kid" value can be used to match a JWS or JWE "kid" Header Parameter value.

3.5. "x5u" (X.509 URL) Parameter

The "x5u" (X.509 URL) member is a URI [RFC3986] that refers to a resource for an X.509 public key certificate or certificate chain [RFC5280]. The identified resource MUST provide a representation of the certificate or certificate chain that conforms to RFC 5280 [RFC5280] in PEM encoded form [RFC1421]. The key in the first

certificate MUST match the bare public key represented by other members of the JWK. The protocol used to acquire the resource MUST provide integrity protection; an HTTP GET request to retrieve the certificate MUST use TLS [RFC2818] [RFC5246]; the identity of the server MUST be validated, as per Section 3.1 of HTTP Over TLS [RFC2818]. Use of this member is OPTIONAL.

3.6. "x5t" (X.509 Certificate SHA-1 Thumbprint) Parameter

The "x5t" (X.509 Certificate SHA-1 Thumbprint) member is a base64url encoded SHA-1 thumbprint (a.k.a. digest) of the DER encoding of an X.509 certificate [RFC5280]. The key in the certificate MUST match the bare public key represented by other members of the JWK. Use of this member is OPTIONAL.

If, in the future, certificate thumbprints need to be computed using hash functions other than SHA-1, it is suggested that additional related JWK parameters be defined for that purpose. For example, it is suggested that a new "x5t#S256" (X.509 Certificate Thumbprint using SHA-256) JWK parameter could be defined by registering it in the IANA JSON Web Key Parameters registry defined in Section 7.1.

3.7. "x5c" (X.509 Certificate Chain) Parameter

The "x5c" (X.509 Certificate Chain) member contains a chain of one or more PKIX certificates [RFC5280]. The certificate chain is represented as a JSON array of certificate value strings. Each string in the array is a base64 encoded ([RFC4648] Section 4 -- not base64url encoded) DER [ITU.X690.1994] PKIX certificate value. The PKIX certificate containing the key value MUST be the first certificate. This MAY be followed by additional certificates, with each subsequent certificate being the one used to certify the previous one. The key in the first certificate MUST match the bare public key represented by other members of the JWK. Use of this member is OPTIONAL.

4. JSON Web Key Set (JWK Set) Format

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

The member names within a JWK Set MUST be unique; recipients MUST either reject JWK Sets with duplicate member names or use a JSON parser that returns only the lexically last duplicate member name, as specified in Section 15.12 (The JSON Object) of ECMAScript 5.1 [ECMAScript].

Additional members MAY be present in the JWK Set. If not understood by implementations encountering them, they MUST be ignored. Parameters for representing additional properties of JWK Sets SHOULD either be registered in the IANA JSON Web Key Set Parameters registry defined in Section 7.3 or be a value that contains a Collision Resistant Name.

Implementations SHOULD ignore JWKs within a JWK Set that use "kty" (key type) values that are not understood by them.

4.1. "keys" Parameter

The value of the "keys" member is an array of JWK values. By default, the order of the JWK values within the array does not imply an order of preference among them, although applications of JWK Sets can choose to assign a meaning to the order for their purposes, if desired. Use of this member is REQUIRED.

5. String Comparison Rules

The string comparison rules for this specification are the same as those defined in Section 5.3 of [JWS].

6. Encrypted JWK and Encrypted JWK Set Formats

JWKs containing non-public key material will need to be encrypted in some contexts to prevent the disclosure of private or symmetric key values to unintended parties. The use of an Encrypted JWK, which is a JWE with a JWK as its plaintext value, is RECOMMENDED for this purpose. The processing of Encrypted JWKs is identical to the processing of other JWEs. A "cty" (content type) Header Parameter value of "jwk+json" MUST be used to indicate that the content of the JWE is a JWK, unless the application knows that the encrypted content is a JWK by another means or convention.

JWK Sets containing non-public key material will similarly need to be encrypted. The use of an Encrypted JWK Set, which is a JWE with a JWK Set as its plaintext value, is RECOMMENDED for this purpose. The processing of Encrypted JWK Sets is identical to the processing of other JWEs. A "cty" (content type) Header Parameter value of "jwk-set+json" MUST be used to indicate that the content of the JWE is a JWK Set, unless the application knows that the encrypted content is a JWK Set by another means or convention.

See Appendix C for an example encrypted JWK.

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 Key Parameters Registry

This specification establishes the IANA JSON Web Key Parameters registry for JWK parameter names. The registry records the parameter

name, the key type(s) that the parameter is used with, and a reference to the specification that defines it. It also records whether the parameter conveys public or private information. This specification registers the parameter names defined in Section 3. The same JWK parameter name may be registered multiple times, provided that duplicate parameter registrations are only for key type specific JWK parameters; in this case, the meaning of the duplicate parameter name is disambiguated by the "kty" value of the JWK containing it.

7.1.1. Registration Template

Parameter 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.

Used with "kty" Value(s):

The key type parameter value(s) that the parameter name is to be used with, or the value "*" if the parameter value is used with all key types.

Parameter Information Class:

Registers whether the parameter conveys public or private information. Its value must be one the words Public or Private.

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

- o Parameter Name: "kty"
- o Used with "kty" Value(s): *

- o Parameter Information Class: Public
- o Change Controller: IESG
- o Specification Document(s): Section 3.1 of [[this document]]

- o Parameter Name: "use"
- o Used with "kty" Value(s): *
- o Parameter Information Class: Public
- o Change Controller: IESG
- o Specification Document(s): Section 3.2 of [[this document]]

- o Parameter Name: "alg"
- o Used with "kty" Value(s): *
- o Parameter Information Class: Public
- o Change Controller: IESG
- o Specification Document(s): Section 3.3 of [[this document]]

- o Parameter Name: "kid"
- o Used with "kty" Value(s): *
- o Parameter Information Class: Public
- o Change Controller: IESG
- o Specification Document(s): Section 3.4 of [[this document]]

- o Parameter Name: "x5u"
- o Used with "kty" Value(s): *
- o Parameter Information Class: Public
- o Change Controller: IESG
- o Specification Document(s): Section 3.5 of [[this document]]

- o Parameter Name: "x5t"
- o Used with "kty" Value(s): *
- o Parameter Information Class: Public
- o Change Controller: IESG
- o Specification Document(s): Section 3.6 of [[this document]]

- o Parameter Name: "x5c"
- o Used with "kty" Value(s): *
- o Parameter Information Class: Public
- o Change Controller: IESG
- o Specification Document(s): Section 3.7 of [[this document]]

7.2. JSON Web Key Use Registry

This specification establishes the IANA JSON Web Key Use registry for JWK "use" member values. The registry records the key use value and a reference to the specification that defines it. This specification registers the parameter names defined in Section 3.2.

7.2.1. Registration Template

Use Member 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.

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.

7.2.2. Initial Registry Contents

- o Use Member Value: "sig"
- o Change Controller: IESG
- o Specification Document(s): Section 3.2 of [[this document]]

- o Use Member Value: "enc"
- o Change Controller: IESG
- o Specification Document(s): Section 3.2 of [[this document]]

7.3. JSON Web Key Set Parameters Registry

This specification establishes the IANA JSON Web Key Set Parameters registry for JWK Set parameter names. The registry records the parameter name and a reference to the specification that defines it. This specification registers the parameter names defined in Section 4.

7.3.1. Registration Template

Parameter 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.

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

- o Parameter Name: "keys"
- o Change Controller: IESG
- o Specification Document(s): Section 4.1 of [[this document]]

7.4. Media Type Registration

7.4.1. Registry Contents

This specification registers the "application/jwk+json" and "application/jwk-set+json" Media Types [RFC2046] in the MIME Media Types registry [IANA.MediaTypes], which can be used to indicate, respectively, that the content is a JWK or a JWK Set.

- o Type Name: application
- o Subtype Name: jwk+json
- o Required Parameters: n/a
- o Optional Parameters: n/a
- o Encoding considerations: 8bit; application/jwk+json values are represented as JSON object; UTF-8 encoding SHOULD be employed for the JSON object.
- o Security Considerations: See the Security Considerations section of [[this document]]
- o Interoperability Considerations: n/a
- o Published Specification: [[this document]]
- o Applications that use this media type: TBD
- o Additional Information: Magic number(s): n/a, File extension(s): n/a, Macintosh file type code(s): n/a
- o Person & email address to contact for further information: Michael B. Jones, mbj@microsoft.com

- o Intended Usage: COMMON
- o Restrictions on Usage: none
- o Author: Michael B. Jones, mbj@microsoft.com
- o Change Controller: IESG

- o Type Name: application
- o Subtype Name: jwk-set+json
- o Required Parameters: n/a
- o Optional Parameters: n/a
- o Encoding considerations: 8bit; application/jwk-set+json values are represented as a JSON Object; UTF-8 encoding SHOULD be employed for the JSON object.
- o Security Considerations: See the Security Considerations section of [[this document]]
- o Interoperability Considerations: n/a
- o Published Specification: [[this document]]
- o Applications that use this media type: TBD
- o Additional Information: Magic number(s): n/a, File extension(s): n/a, Macintosh file type code(s): n/a
- o Person & email address to contact for further information: Michael B. Jones, mbj@microsoft.com
- o Intended Usage: COMMON
- o Restrictions on Usage: none
- o Author: Michael B. Jones, mbj@microsoft.com
- o Change Controller: IESG

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.

A key is no more trustworthy than the method by which it was received.

Private and symmetric keys MUST be protected from disclosure to unintended parties. One recommended means of doing so is to encrypt JWKs or JWK Sets containing them by using the JWK or JWK Set value as the plaintext of a JWE.

The security considerations in RFC 3447 [RFC3447] and RFC 6030 [RFC6030] about protecting private and symmetric keys also apply to this specification.

The security considerations in XML DSIG 2.0 [W3C.CR-xmlsig-core2-20120124], about key representations also apply to this specification, other than those that are XML specific.

The TLS Requirements in [JWS] also apply to this specification.

9. References

9.1. Normative References

[ECMAScript]

Ecma International, "ECMAScript Language Specification, 5.1 Edition", ECMA 262, June 2011.

[IANA.MediaType]

Internet Assigned Numbers Authority (IANA), "MIME Media Types", 2005.

[ITU.X690.1994]

International Telecommunications Union, "Information Technology - ASN.1 encoding rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER)", ITU-T Recommendation X.690, 1994.

[JWA]

Jones, M., "JSON Web Algorithms (JWA)", draft-ietf-jose-json-web-algorithms (work in progress), October 2013.

[JWE]

Jones, M., Rescorla, E., and J. Hildebrand, "JSON Web Encryption (JWE)", draft-ietf-jose-json-web-encryption (work in progress), October 2013.

[JWS]

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

[RFC1421]

Linn, J., "Privacy Enhancement for Internet Electronic Mail: Part I: Message Encryption and Authentication Procedures", RFC 1421, February 1993.

[RFC2046]

Freed, N. and N. Borenstein, "Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types", RFC 2046, November 1996.

[RFC2119]

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

- [RFC2818] Rescorla, E., "HTTP Over TLS", RFC 2818, May 2000.
- [RFC3629] Yergeau, F., "UTF-8, a transformation format of ISO 10646", STD 63, RFC 3629, November 2003.
- [RFC3986] Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax", STD 66, RFC 3986, January 2005.
- [RFC4627] Crockford, D., "The application/json Media Type for JavaScript Object Notation (JSON)", RFC 4627, July 2006.
- [RFC4648] Josefsson, S., "The Base16, Base32, and Base64 Data Encodings", RFC 4648, October 2006.
- [RFC5226] Narten, T. and H. Alvestrand, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 5226, May 2008.
- [RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, August 2008.
- [RFC5280] Cooper, D., Santesson, S., Farrell, S., Boeyen, S., Housley, R., and W. Polk, "Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile", RFC 5280, May 2008.
- [USASCII] American National Standards Institute, "Coded Character Set -- 7-bit American Standard Code for Information Interchange", ANSI X3.4, 1986.
- [W3C.CR-xmlsig-core2-20120124]
Eastlake, D., Reagle, J., Yiu, K., Solo, D., Datta, P., Hirsch, F., Cantor, S., and T. Roessler, "XML Signature Syntax and Processing Version 2.0", World Wide Web Consortium CR CR-xmlsig-core2-20120124, January 2012, <<http://www.w3.org/TR/2012/CR-xmlsig-core2-20120124>>.

9.2. Informative References

- [MagicSignatures]
Panzer (editor), J., Laurie, B., and D. Balfanz, "Magic Signatures", January 2011.
- [RFC3447] Jonsson, J. and B. Kaliski, "Public-Key Cryptography Standards (PKCS) #1: RSA Cryptography Specifications Version 2.1", RFC 3447, February 2003.

[RFC6030] Hoyer, P., Pei, M., and S. Machani, "Portable Symmetric Key Container (PSKC)", RFC 6030, October 2010.

Appendix A. Example JSON Web Key Sets

A.1. Example Public Keys

The following example JWK Set contains two public keys represented as JWKs: one using an Elliptic Curve algorithm and a second one using an RSA algorithm. The first specifies that the key is to be used for encryption. The second specifies that the key is to be used with the "RS256" algorithm. Both provide a Key ID for key matching purposes. In both cases, integers are represented using the base64url encoding of their big endian representations. (Long lines are broken for display purposes only.)

```
{ "keys":
  [
    { "kty": "EC",
      "crv": "P-256",
      "x": "MKBCtN1cKUSDiillySs3526iDZ8AiTo7Tu6KPAqv7D4",
      "y": "4Et16SRW2YiLUrN5vfvVHuhp7x8PxltmWWlbbM4IFyM",
      "use": "enc",
      "kid": "1" },
    { "kty": "RSA",
      "n": "0vx7agoebGcQSuuPiLJXZptN9nndrQmbXEps2aiAFbWhM78LhWx
4cbbfAAAtVT86zwu1RK7aPFFxuhDR1L6tSoc_BJECPEbWKRxbZCiFV4n3oknjhMs
tn64tZ_2W-5JsgY4Hc5n9yBXArwl93lqt7_RN5w6Cf0h4QyQ5v-65YGjQR0_FDW2
QvzqY368QQMicAtaSqzs8KJZgnYb9c7d0zgdAZHzu6QMqvRL5hajrnln91CbOpbI
SD08qNLyrdkt-bFTWhAI4vMQFh6WeZu0fM4lFd2NcRwr3XPksINHaQ-G_xBniIqbw0LsljF44-csFCur-kEgU8awapJzKnqDKgw",
      "e": "AQAB",
      "alg": "RS256",
      "kid": "2011-04-29" }
  ]
}
```

A.2. Example Private Keys

The following example JWK Set contains two keys represented as JWKs containing both public and private key values: one using an Elliptic Curve algorithm and a second one using an RSA algorithm. This example extends the example in the previous section, adding private key values. (Line breaks are for display purposes only.)

```

{ "keys":
  [
    { "kty": "EC",
      "crv": "P-256",
      "x": "MKBCTNIcKUSDiillySs3526iDZ8AiTo7Tu6KPAqv7D4",
      "y": "4Et16SRW2YiLUrN5vfvVHuhp7x8PxltmWWlbbM4IFyM",
      "d": "870MB6gfuTJ4HtUnUvYMyJpr5eUZNP4Bk43bVdj3eAE",
      "use": "enc",
      "kid": "1" },

    { "kty": "RSA",
      "n": "0vx7agoebGcQSuuPiLjXZptN9nndrQmbXEps2aiAFbWhM78LhWx4
cbbfAAAtVT86zwulRK7aPFFxuhDR1L6tSoc_BJECPEbWKRXjBZCiFV4n3oknjhMst
n64tZ_2W-5JsGY4Hc5n9yBXArwl93lqt7_RN5w6Cf0h4QyQ5v-65YGjQR0_FDW2Q
vzqY368QQMicAtaSqzs8KJZgnYb9c7d0zgdAZHzu6QMqvRL5hajrnln91CbOpbIS
D08qNlYrdkt-bFTWhAI4vMQFh6WeZu0fm4lFd2NcRwr3XPksINHaQ-G_xBniIqbw
0LsljF44-csFCur-kEgU8awapJzKnqDKgw",
      "e": "AQAB",
      "d": "X4cTteJY_gn4FYPsXB8rdXix5vwsg1FLN5E3EaG6RJoVH-HLLKD9
M7dx5oo7GURknchnrRweUkC7hT5fJLM0WbFAKNLWY2vv7B6NqXSzUvxtT0_YSfqi j
wp3RTzlBaCxWp4doFk5N2o8Gy_nHNKroADiKJ46pRUohsXywbReAdYaMwFs9tv8d
_cPVY3i07a3t8MN6TNwm0dSawm9v47UiCl3Sk5ZiG7xojPLu4sbglU2jx4IBTNBz
nbJSzFHK66jT8bgkuqsk0GjskDJk19Z4qwjbsnn4j2WBii3RL-Us2lGVkY8fkFz
melz0HbIkfz0Y6mqnOYtqc0X4jfcKoAC8Q",
      "p": "83i-7IvMGXoMXCskv73TKr8637Fi07Z27zv8oj6pbWUQyLPQBQxtPV
nwd20R-60eTDmD2ujnMt5PoqMrm8RfmNhVWDtjjMmCMjOpSXicFHj7XOuVIYQyqV
WlWEh6dN36GVZYk93N8Bc9vY41xy8B9RzzOGVQzXvNEvn700nVbfs",
      "q": "3dfOR9cuYq-0S-mkFLzgItgMEffzB2q3hWehMuG0oCuqnb3vobLyum
qjVZQ0ldIrdwgTnCdpYzBcOfW5r370AFXjiWft_NGEiovonizhKpo9VVS78TzFgx
kIdrecRezsZ-1kYd_slqDbxtkDEgfAITAG9LUnADun4vIcb6yelxk",
      "dp": "G4sPXkc6Ya9y8oJW9_ILj4xuppu0lzi_H7VTkS8xj5SdX3coE0oim
YwxIi2emTAue0UOa5dpgFGyBJ4c8tQ2VF402XRugKDTP8akYhFo5tAA77Qe_Nmtu
YZc3C3m3I24G2GvR5sSDxUyAN2zq8Lfn9EUms6rY3Ob8YeikKtiBj0",
      "dq": "s91AH9fggBsoFR8Oac2R_E2gw282rT2kGOAhvIl1ETE1efrA6huUU
vMfBcMpn8lqeW6vzznYY5SSQF7pMdc_agI3nG8IbplBUB0JUiraRNqUfLhcQb_d9
GF4Dh7e74WbRsobRonujTYNlxCaP6TO61jvWrX-L18txXw494Q_cgk",
      "qi": "GyM_p6JrXySizltoFgKbWV-JdI3jQ4ypu9rbMWx3rQJBfmt0FoYzg
UIZEVFEcOqwemRN81zoDAaa-Bk0KWNGDjJHZDdDmFhW3AN71I-puxk_mHZGJ11rx
yR8O55XLSe3SPmRfKwZI6yU24ZxvQKFYItldldUKGzO6Ia6zTKhAVRU",
      "alg": "RS256",
      "kid": "2011-04-29" }
  ]
}

```

A.3. Example Symmetric Keys

The following example JWK Set contains two symmetric keys represented as JWKs: one designated as being for use with the AES Key Wrap algorithm and a second one that is an HMAC key. (Line breaks are for display purposes only.)

```
{ "keys":  
  [  
    { "kty": "oct",  
      "alg": "A128KW",  
      "k": "GawggufyGrWKav7AX4VKUg" },  
  
    { "kty": "oct",  
      "k": "AyMlSysPpbyDfgZld3umjlqzKObwVMkoqQ-EstJQLr_T-1qS0gZH75  
aKtMN3Yj0iPS4hcgUuTwjAzZr1Z9CAow",  
      "kid": "HMAC key used in JWS A.1 example" }  
  ]  
}
```

Appendix B. Example Use of "x5c" (X.509 Certificate Chain) Parameter

The following is an example of a JWK with a RSA signing key represented both as a bare public key and as an X.509 certificate using the "x5c" parameter:

```
{ "kty": "RSA",
  "use": "sig",
  "kid": "1b94c",
  "n": "vrjOfz9Ccdgx5nQudyhdoR17V-IubWMeOZCwX_jj0hgAsz2J_pqYW08
  PLbK_PdiVGKPrqzmDI7sA25VENHuluCLNwBuUiC011_-7dYbsr4iJmG0Q
  u2j8DsVyTlazpJC_NG84Ty5KKthuCaPod7iI7w0LK9orSMhBEwwZDCxTWq4a
  YWAchc8t-emd9qOvWtVMDC2BXksRngh6X5bUYLy6AyHKvj-nUylwgzjYQDwH
  MTplCoLtU-o-8SNnZltnRoGE9uJkBLdh5gFENabWnU5mlZqZPdws-qo-meMv
  VfJb6jJVWRpl2SutCnYG2C32qvbWbjZ_jBPD5eunqsIo1vQ",
  "e": "AQAB",
  "x5c":
  [ "MIIDQjCCAiqgAwIBAgIGATz/FuLiMA0GCSqGSIb3DQEBBQUAMGIXCzAJB
  gNVBAYTA1VTMQswCQYDVQQIEwJDTzEPMA0GA1UEBxMGRGVudmVyMRwwGgYD
  VQQKEzNQaW5nIElkZW50aXR5IENvcnAuMRcwFQYDVQQDEw5CcmlhbiBDYWl
  wYmVsbDAeFw0xMzAyMjE5MTVaFw0xODA4MTQyMjE5MTVaMGIXCzAJBg
  NVBAYTA1VTMQswCQYDVQQIEwJDTzEPMA0GA1UEBxMGRGVudmVyMRwwGgYD
  VQQKEzNQaW5nIElkZW50aXR5IENvcnAuMRcwFQYDVQQDEw5CcmlhbiBDYWl
  wYmVsbDCCASIwDQYJKoZIhvcNAQEBBQADggEPADCCAQoCggEBAL64zn8/QnH
  YMeZ0LncOXAEdelfiLmljHjmQsF/449IYALM9if6amFtPDy2yvv3YlRij66
  s5gyLCyO7ANuVRJxlNbgizcAblIgjtdf/u3WG7K+IiZhtELto/A7Fck9Ws6
  SQvzRvOE8uSirYbgmj6He4iO8NCyvaK0jIQRMMGQwsU1quGmFgHIXPLfnpn
  fajrlrVTAwtgV5LEZ4Iel+WlGC8ugMhyr4/plMtcIM42EA8BzE6ZQqC7VPq
  PvEjz2dbZkaBhPbiZAS3YeYBRDwmlp1OZtWamT3cEvqqPpnjLlXyW+oyVVk
  aZdkllQp2Btgt9qr2lm42f4wTw+Xrp6rCKNb0CAwEAATANBgkqhkiG9w0BA
  QUFAAOCAQEAh8zGlfsIcI0o3rYDPBB07aXNswb4ECNKG0CETTUxmXl9KUL
  +9gGlqCz5iWLOgWsnrcKcY0vXPG9Jlr9AqBNTqNgHq2G03X09266X5CpOe1
  zFo+Owblzxt3PehFdfQJ6l0CDLEaS9V9Rqp17hCyybEpOGVwe8fnk+fbEL
  2Bo3UPGrpsHzUoaGpDftmWssZkhpbJKVMJyf/RuP2SmmaIzmnw9JiSlYhzo
  4tpzd5rFXhjRbg4zW9C+2qok+2+qDM1iJ684gPHMIY8aLWrdgQTxkumGmTq
  gawR+N5MDtdPTEQ0XfIBc2cJEUyMTY5MPvACWpkA6SdS4xSvdXK3IVfOWA==" ]
}
```

Appendix C. Example Encrypted RSA Private Key

This example encrypts an RSA private key to the recipient using "PBES2-HS256+A128KW" for key encryption and "A128CBC+HS256" for content encryption.

NOTE: Unless otherwise indicated, all line breaks are included solely for readability.

C.1. Plaintext RSA Private Key

The following RSA key is the plaintext for the encryption operation, formatted as a JWK object:

```
{
  "kty": "RSA",
  "kid": "juliet@capulet.lit",
  "use": "enc",
  "n": "t6Q8PWSi1dkJj9hTP8hNYFlvadM7DflW9mWepOJhJ66w7nyoK1gPNqFMSQRy
    O125Gp-TEkodhWr0iujjHVx7BcV01lS4w5ACGgPrcAd6ZcSR0-Iqom-QFcNP
    8Sjg086MwoqQU_LYywlAGZ21WSdS_PERYGFInnj3QQ108Yns5jCtLCRwLHL0
    PblfEv45AuRIuUfVcPySBWYnDyGxvjYGDSM-AqWS9zIQ2ZilgT-GqUmipg0X
    OC0Cc20rgLe2ymLHjpHciCKVAbY5-L32-lSeZ0-Os6U15_aXrk9Gw8cPUaX1
    _I8sLGuSiVdt3C_Fn2PZ3Z8i744FPFGGcG1qs2Wz-Q",
  "e": "AQAB",
  "d": "GRtBIQmhOZtyszfGkdG4u_N-R_mZGU_9k7JQ_jn1DnfTuMdSNprTeaSTyWfS
    NkuaAwnOEbIQVylIQbWVV25NY3ybc_IhUJtfri7bAXYEReWaCl3hdlPKXy9U
    vqPYGR0kIXTQRqns-dVJ7jah1I7LyckrpTmrM8dWBo4_PMaenNnPiQgO0xnu
    ToxutRZJfJvG4Ox4ka3GORQd9CsCZ2vsUDmsXOfUEN0yMqADC6p1M3h33tsu
    rY15k9qMSpG9OX_IJAXmxzAh_tWiZOWk2K4yxH9ts3LqlyX8ClEWmerDkK2a
    hecG85-oLKQt5VEpWHKmjiOi_gJSdSgqcN96X52esAQ",
  "p": "2rnSOV4hKSN8sS4CgcQHFbs08XboFDqKum3sc4h3GRxrTmQdl1ZK9uw-PIHf
    QP0FkxXVrx-WE-ZEbrqivH_2iCLUS7wAl6XvARTlKkIaUxPPSYB9yk31s0Q8
    UK96E3_OrADAYtAJs-M3JxCLfNgqh56HDnETTQhH3rCT5T3yJws",
  "q": "lu_RiFDP7LBYh3N4GXLt9OpSKYP0uQZyiaZwBtOCBNJgQxaJl0RWjsZu0c6I
    edis4S7B_coSKB0Kj9PaPaBzg-IySRvvcQuPamQu66riMhjVtG6TlV8CLCYK
    rY152ziqK0E_ym2QnkwsUX7eYTB7LbAHRK9GqocDE5B0f808I4s",
  "dp": "KkMTWqBUefVwZ2_DbjlpPQqyHSHjj90L5x_MOzqYAJMcLMZtbUtwKqvVDq3
    tbEo3ZiCohbDtt6SbfmWzggabpQxNxuBpoOOf_a_HgMXK_lhqigI4y_kqS1w
    Y52IwJUn5rgRrJ-yYolh4lKR-vz2pYhEAeYrhttWtxVqLCRviD6c",
  "dq": "AvfS0-gRxvn0bwJoMSnFxyCk1WnuEjQFluMGfwGitQBWtfZ1Er7t1xDkbN9
    GQTB9yqpDoYaN06H7CFtrkxhJIBQaj6nkF5KKS3TQtQ5qCzkOkmxIe3KRbBy
    mXxkb5qwUpX5ELD5xFc6FeiafWYY63TmmEAU_lRFCOJ3xDea-ots",
  "qi": "lSQi-w9CpyUREMerPlRsBLk7wNtOvs5EQpPqmuMvqW57NBUCzScEoPwmUqq
    abu9V0-Py4dQ57_bapoKRulR90bvuFnU63SHWEFglZQvJDMaAvmj4sm-Fp0o
    Yu_neotgQ0hzbI5gry7ajdYy9-2lNx_76aBZoOUu9HCJ-UsfSOI8"
}
```

The octets representing the Plaintext are:

```
[ 123, 34, 107, 116, 121, 34, 58, 34, 82, 83, 65, 34, 44, 34, 107,
  105, 100, 34, 58, 34, 106, 117, 108, 105, 101, 116, 64, 99, 97, 112,
  117, 108, 101, 116, 46, 108, 105, 116, 34, 44, 34, 117, 115, 101, 34,
  58, 34, 101, 110, 99, 34, 44, 34, 110, 34, 58, 34, 116, 54, 81, 56,
  80, 87, 83, 105, 49, 100, 107, 74, 106, 57, 104, 84, 80, 56, 104, 78,
  89, 70, 108, 118, 97, 100, 77, 55, 68, 102, 108, 87, 57, 109, 87,
  101, 112, 79, 74, 104, 74, 54, 54, 119, 55, 110, 121, 111, 75, 49,
```

103, 80, 78, 113, 70, 77, 83, 81, 82, 121, 79, 49, 50, 53, 71, 112, 45, 84, 69, 107, 111, 100, 104, 87, 114, 48, 105, 117, 106, 106, 72, 86, 120, 55, 66, 99, 86, 48, 108, 108, 83, 52, 119, 53, 65, 67, 71, 103, 80, 114, 99, 65, 100, 54, 90, 99, 83, 82, 48, 45, 73, 113, 111, 109, 45, 81, 70, 99, 78, 80, 56, 83, 106, 103, 48, 56, 54, 77, 119, 111, 113, 81, 85, 95, 76, 89, 121, 119, 108, 65, 71, 90, 50, 49, 87, 83, 100, 83, 95, 80, 69, 82, 121, 71, 70, 105, 78, 110, 106, 51, 81, 81, 108, 79, 56, 89, 110, 115, 53, 106, 67, 116, 76, 67, 82, 119, 76, 72, 76, 48, 80, 98, 49, 102, 69, 118, 52, 53, 65, 117, 82, 73, 117, 85, 102, 86, 99, 80, 121, 83, 66, 87, 89, 110, 68, 121, 71, 120, 118, 106, 89, 71, 68, 83, 77, 45, 65, 113, 87, 83, 57, 122, 73, 81, 50, 90, 105, 108, 103, 84, 45, 71, 113, 85, 109, 105, 112, 103, 48, 88, 79, 67, 48, 67, 99, 50, 48, 114, 103, 76, 101, 50, 121, 109, 76, 72, 106, 112, 72, 99, 105, 67, 75, 86, 65, 98, 89, 53, 45, 76, 51, 50, 45, 108, 83, 101, 90, 79, 45, 79, 115, 54, 85, 49, 53, 95, 97, 88, 114, 107, 57, 71, 119, 56, 99, 80, 85, 97, 88, 49, 95, 73, 56, 115, 76, 71, 117, 83, 105, 86, 100, 116, 51, 67, 95, 70, 110, 50, 80, 90, 51, 90, 56, 105, 55, 52, 52, 70, 80, 70, 71, 71, 99, 71, 49, 113, 115, 50, 87, 122, 45, 81, 34, 44, 34, 101, 34, 58, 34, 65, 81, 65, 66, 34, 44, 34, 100, 34, 58, 34, 71, 82, 116, 98, 73, 81, 109, 104, 79, 90, 116, 121, 115, 122, 102, 103, 75, 100, 103, 52, 117, 95, 78, 45, 82, 95, 109, 90, 71, 85, 95, 57, 107, 55, 74, 81, 95, 106, 110, 49, 68, 110, 102, 84, 117, 77, 100, 83, 78, 112, 114, 84, 101, 97, 83, 84, 121, 87, 102, 83, 78, 107, 117, 97, 65, 119, 110, 79, 69, 98, 73, 81, 86, 121, 49, 73, 81, 98, 87, 86, 86, 50, 53, 78, 89, 51, 121, 98, 99, 95, 73, 104, 85, 74, 116, 102, 114, 105, 55, 98, 65, 88, 89, 69, 82, 101, 87, 97, 67, 108, 51, 104, 100, 108, 80, 75, 88, 121, 57, 85, 118, 113, 80, 89, 71, 82, 48, 107, 73, 88, 84, 81, 82, 113, 110, 115, 45, 100, 86, 74, 55, 106, 97, 104, 108, 73, 55, 76, 121, 99, 107, 114, 112, 84, 109, 114, 77, 56, 100, 87, 66, 111, 52, 95, 80, 77, 97, 101, 110, 78, 110, 80, 105, 81, 103, 79, 48, 120, 110, 117, 84, 111, 120, 117, 116, 82, 90, 74, 102, 74, 118, 71, 52, 79, 120, 52, 107, 97, 51, 71, 79, 82, 81, 100, 57, 67, 115, 67, 90, 50, 118, 115, 85, 68, 109, 115, 88, 79, 102, 85, 69, 78, 79, 121, 77, 113, 65, 68, 67, 54, 112, 49, 77, 51, 104, 51, 51, 116, 115, 117, 114, 89, 49, 53, 107, 57, 113, 77, 83, 112, 71, 57, 79, 88, 95, 73, 74, 65, 88, 109, 120, 122, 65, 104, 95, 116, 87, 105, 90, 79, 119, 107, 50, 75, 52, 121, 120, 72, 57, 116, 83, 51, 76, 113, 49, 121, 88, 56, 67, 49, 69, 87, 109, 101, 82, 68, 107, 75, 50, 97, 104, 101, 99, 71, 56, 53, 45, 111, 76, 75, 81, 116, 53, 86, 69, 112, 87, 72, 75, 109, 106, 79, 105, 95, 103, 74, 83, 100, 83, 103, 113, 99, 78, 57, 54, 88, 53, 50, 101, 115, 65, 81, 34, 44, 34, 112, 34, 58, 34, 50, 114, 110, 83, 79, 86, 52, 104, 75, 83, 78, 56, 115, 83, 52, 67, 103, 99, 81, 72, 70, 98, 115, 48, 56, 88, 98, 111, 70, 68, 113, 75, 117, 109, 51, 115, 99, 52, 104, 51, 71, 82, 120, 114, 84, 109, 81, 100, 108, 49, 90, 75, 57, 117, 119, 45, 80, 73, 72, 102, 81, 80, 48, 70, 107, 120, 88, 86, 114, 120, 45, 87, 69, 45, 90, 69, 98, 114, 113, 105, 118, 72, 95, 50, 105, 67, 76, 85, 83, 55, 119, 65, 108, 54, 88, 118, 65, 82, 116, 49, 75,

107, 73, 97, 85, 120, 80, 80, 83, 89, 66, 57, 121, 107, 51, 49, 115, 48, 81, 56, 85, 75, 57, 54, 69, 51, 95, 79, 114, 65, 68, 65, 89, 116, 65, 74, 115, 45, 77, 51, 74, 120, 67, 76, 102, 78, 103, 113, 104, 53, 54, 72, 68, 110, 69, 84, 84, 81, 104, 72, 51, 114, 67, 84, 53, 84, 51, 121, 74, 119, 115, 34, 44, 34, 113, 34, 58, 34, 49, 117, 95, 82, 105, 70, 68, 80, 55, 76, 66, 89, 104, 51, 78, 52, 71, 88, 76, 84, 57, 79, 112, 83, 75, 89, 80, 48, 117, 81, 90, 121, 105, 97, 90, 119, 66, 116, 79, 67, 66, 78, 74, 103, 81, 120, 97, 106, 49, 48, 82, 87, 106, 115, 90, 117, 48, 99, 54, 73, 101, 100, 105, 115, 52, 83, 55, 66, 95, 99, 111, 83, 75, 66, 48, 75, 106, 57, 80, 97, 80, 97, 66, 122, 103, 45, 73, 121, 83, 82, 118, 118, 99, 81, 117, 80, 97, 109, 81, 117, 54, 54, 114, 105, 77, 104, 106, 86, 116, 71, 54, 84, 108, 86, 56, 67, 76, 67, 89, 75, 114, 89, 108, 53, 50, 122, 105, 113, 75, 48, 69, 95, 121, 109, 50, 81, 110, 107, 119, 115, 85, 88, 55, 101, 89, 84, 66, 55, 76, 98, 65, 72, 82, 75, 57, 71, 113, 111, 99, 68, 69, 53, 66, 48, 102, 56, 48, 56, 73, 52, 115, 34, 44, 34, 100, 112, 34, 58, 34, 75, 107, 77, 84, 87, 113, 66, 85, 101, 102, 86, 119, 90, 50, 95, 68, 98, 106, 49, 112, 80, 81, 113, 121, 72, 83, 72, 106, 106, 57, 48, 76, 53, 120, 95, 77, 79, 122, 113, 89, 65, 74, 77, 99, 76, 77, 90, 116, 98, 85, 116, 119, 75, 113, 118, 86, 68, 113, 51, 116, 98, 69, 111, 51, 90, 73, 99, 111, 104, 98, 68, 116, 116, 54, 83, 98, 102, 109, 87, 122, 103, 103, 97, 98, 112, 81, 120, 78, 120, 117, 66, 112, 111, 79, 79, 102, 95, 97, 95, 72, 103, 77, 88, 75, 95, 108, 104, 113, 105, 103, 73, 52, 121, 95, 107, 113, 83, 49, 119, 89, 53, 50, 73, 119, 106, 85, 110, 53, 114, 103, 82, 114, 74, 45, 121, 89, 111, 49, 104, 52, 49, 75, 82, 45, 118, 122, 50, 112, 89, 104, 69, 65, 101, 89, 114, 104, 116, 116, 87, 116, 120, 86, 113, 76, 67, 82, 86, 105, 68, 54, 99, 34, 44, 34, 100, 113, 34, 58, 34, 65, 118, 102, 83, 48, 45, 103, 82, 120, 118, 110, 48, 98, 119, 74, 111, 77, 83, 110, 70, 120, 89, 99, 75, 49, 87, 110, 117, 69, 106, 81, 70, 108, 117, 77, 71, 102, 119, 71, 105, 116, 81, 66, 87, 116, 102, 90, 49, 69, 114, 55, 116, 49, 120, 68, 107, 98, 78, 57, 71, 81, 84, 66, 57, 121, 113, 112, 68, 111, 89, 97, 78, 48, 54, 72, 55, 67, 70, 116, 114, 107, 120, 104, 74, 73, 66, 81, 97, 106, 54, 110, 107, 70, 53, 75, 75, 83, 51, 84, 81, 116, 81, 53, 113, 67, 122, 107, 79, 107, 109, 120, 73, 101, 51, 75, 82, 98, 66, 121, 109, 88, 120, 107, 98, 53, 113, 119, 85, 112, 88, 53, 69, 76, 68, 53, 120, 70, 99, 54, 70, 101, 105, 97, 102, 87, 89, 89, 54, 51, 84, 109, 109, 69, 65, 117, 95, 108, 82, 70, 67, 79, 74, 51, 120, 68, 101, 97, 45, 111, 116, 115, 34, 44, 34, 113, 105, 34, 58, 34, 108, 83, 81, 105, 45, 119, 57, 67, 112, 121, 85, 82, 101, 77, 69, 114, 80, 49, 82, 115, 66, 76, 107, 55, 119, 78, 116, 79, 118, 115, 53, 69, 81, 112, 80, 113, 109, 117, 77, 118, 113, 87, 53, 55, 78, 66, 85, 99, 122, 83, 99, 69, 111, 80, 119, 109, 85, 113, 113, 97, 98, 117, 57, 86, 48, 45, 80, 121, 52, 100, 81, 53, 55, 95, 98, 97, 112, 111, 75, 82, 117, 49, 82, 57, 48, 98, 118, 117, 70, 110, 85, 54, 51, 83, 72, 87, 69, 70, 103, 108, 90, 81, 118, 74, 68, 77, 101, 65, 118, 109, 106, 52, 115, 109, 45, 70, 112, 48, 111, 89, 117, 95, 110, 101, 111, 116, 103, 81, 48, 104, 122, 98, 73, 53, 103, 114, 121, 55, 97, 106,


```
100, 89, 121, 57, 45, 50, 108, 78, 120, 95, 55, 54, 97, 66, 90, 111,
79, 85, 117, 57, 72, 67, 74, 45, 85, 115, 102, 83, 79, 73, 56, 34,
125 ]
```

C.2. JWE Header

The following example JWE Protected Header declares that:

- o the Content Encryption Key is encrypted to the recipient using the PSE2-HS256+A128KW algorithm to produce the JWE Encrypted Key,
- o the Salt (p2s) is [217, 96, 147, 112, 150, 117, 70, 247, 127, 8, 155, 137, 174, 42, 80, 215],
- o the Iteration Count (p2c) is 4096,
- o the Plaintext is encrypted using the AES_128_CBC_HMAC_SHA_256 algorithm to produce the Ciphertext, and
- o the content type is application/jwk+json.

```
{
  "alg": "PBES2-HS256+A128KW",
  "p2s": "2WCTcJZlRvd_CJuJripQlw",
  "p2c": 4096,
  "enc": "A128CBC-HS256",
  "cty": "jwk+json"
}
```

Encoding this JWE Protected Header as BASE64URL(UTF8(JWE Protected Header)) gives this value:

```
eyJhbGciOiJQQkVtMi1Uzi1lNitBMTI4SlciLCJwMnMiOiIyV0NUY0paMVJ2ZF9DSn
VKcmlwUTF3IiwicDJjIjo0MDk2LCJlbmMiOiJBMTI4Q0JDLUhTMjU2IiwieY3R5Ijoj
andrK2pzb24ifQ
```

C.3. Content Encryption Key (CEK)

Generate a 256 bit random Content Encryption Key (CEK). In this example, the value is:

```
[ 111, 27, 25, 52, 66, 29, 20, 78, 92, 176, 56, 240, 65, 208, 82,
112, 161, 131, 36, 55, 202, 236, 185, 172, 129, 23, 153, 194, 195,
48, 253, 182 ]
```

C.4. Key Encryption

Encrypt the CEK with a shared passphrase using the "PBES2-HS256+A128KW" algorithm and the specified Salt and Iteration Count values to produce the JWE Encrypted Key. This example uses the following passphrase:

Thus from my lips, by yours, my sin is purged.

The octets representing the passphrase are:

```
[ 84, 104, 117, 115, 32, 102, 114, 111, 109, 32, 109, 121, 32, 108,
105, 112, 115, 44, 32, 98, 121, 32, 121, 111, 117, 114, 115, 44, 32,
109, 121, 32, 115, 105, 110, 32, 105, 115, 32, 112, 117, 114, 103,
101, 100, 46 ]
```

The resulting JWE Encrypted Key value is:

```
[ 201, 236, 143, 112, 12, 234, 200, 211, 33, 241, 255, 65, 112, 63,
172, 146, 105, 107, 122, 0, 30, 21, 44, 21, 14, 61, 200, 57, 30, 253,
228, 83, 218, 82, 138, 80, 121, 254, 193, 121 ]
```

Encoding this JWE Encrypted Key as BASE64URL(JWE Encrypted Key) gives this value:

```
yeyPcAzqyNMh8f9BcD-skmlregAeFSwVDj3IOR795FPaUopQef7BeQ
```

C.5. Initialization Vector

Generate a random 128 bit JWE Initialization Vector. In this example, the value is:

```
[ 97, 239, 99, 214, 171, 54, 216, 57, 145, 72, 7, 93, 34, 31, 149,
156 ]
```

Encoding this JWE Initialization Vector as BASE64URL(JWE Initialization Vector) gives this value:

```
Ye9jlqs22DmRSAddIh-VnA
```

C.6. Additional Authenticated Data

Let the Additional Authenticated Data encryption parameter be ASCII(BASE64URL(UTF8(JWE Protected Header))). This value is:

```
[ 123, 34, 97, 108, 103, 34, 58, 34, 80, 66, 69, 83, 50, 45, 72, 83,
50, 53, 54, 43, 65, 49, 50, 56, 75, 87, 34, 44, 34, 112, 50, 115, 34,
58, 34, 50, 87, 67, 84, 99, 74, 90, 49, 82, 118, 100, 95, 67, 74,
```

117, 74, 114, 105, 112, 81, 49, 119, 34, 44, 34, 112, 50, 99, 34, 58, 52, 48, 57, 54, 44, 34, 101, 110, 99, 34, 58, 34, 65, 49, 50, 56, 67, 66, 67, 45, 72, 83, 50, 53, 54, 34, 44, 34, 99, 116, 121, 34, 58, 34, 106, 119, 107, 43, 106, 115, 111, 110, 34, 125]

C.7. Content Encryption

Encrypt the Plaintext with AES_128_CBC_HMAC_SHA_256 using the CEK as the encryption key, the JWE Initialization Vector, and the Additional Authenticated Data value above. The resulting Ciphertext is:

[3, 8, 65, 242, 92, 107, 148, 168, 197, 159, 77, 139, 25, 97, 42, 131, 110, 199, 225, 56, 61, 127, 38, 64, 108, 91, 247, 167, 150, 98, 112, 122, 99, 235, 132, 50, 28, 46, 56, 170, 169, 89, 220, 145, 38, 157, 148, 224, 66, 140, 8, 169, 146, 117, 222, 54, 242, 28, 31, 11, 129, 227, 226, 169, 66, 117, 133, 254, 140, 216, 115, 203, 131, 60, 60, 47, 233, 132, 121, 13, 35, 188, 53, 19, 172, 77, 59, 54, 211, 158, 172, 25, 60, 111, 0, 80, 201, 158, 160, 210, 68, 55, 12, 67, 136, 130, 87, 216, 197, 95, 62, 20, 155, 205, 5, 140, 27, 168, 221, 65, 114, 78, 157, 254, 46, 206, 182, 52, 135, 87, 239, 3, 34, 186, 126, 220, 151, 17, 33, 237, 57, 96, 172, 183, 58, 45, 248, 103, 241, 142, 136, 7, 53, 16, 173, 181, 7, 93, 92, 252, 1, 53, 212, 242, 8, 255, 11, 239, 181, 24, 148, 136, 111, 24, 161, 244, 23, 106, 69, 157, 215, 243, 189, 240, 166, 169, 249, 72, 38, 201, 99, 223, 173, 229, 9, 222, 82, 79, 157, 176, 248, 85, 239, 121, 163, 1, 31, 48, 98, 206, 61, 249, 104, 216, 201, 227, 105, 48, 194, 193, 10, 36, 160, 159, 241, 166, 84, 54, 188, 211, 243, 242, 40, 46, 45, 193, 193, 160, 169, 101, 201, 1, 73, 47, 105, 142, 88, 28, 42, 132, 26, 61, 58, 63, 142, 243, 77, 26, 179, 153, 166, 46, 203, 208, 49, 55, 229, 34, 178, 4, 109, 180, 204, 204, 115, 1, 103, 193, 5, 91, 215, 214, 195, 1, 110, 208, 53, 144, 36, 105, 12, 54, 25, 129, 101, 15, 183, 150, 250, 147, 115, 227, 58, 250, 5, 128, 232, 63, 15, 14, 19, 141, 124, 253, 142, 137, 189, 135, 26, 44, 240, 27, 88, 132, 105, 127, 6, 71, 37, 41, 124, 187, 165, 140, 34, 200, 123, 80, 228, 24, 231, 176, 132, 171, 138, 145, 152, 116, 224, 50, 141, 51, 147, 91, 186, 7, 246, 106, 217, 148, 244, 227, 244, 45, 220, 121, 165, 224, 148, 181, 17, 181, 128, 197, 101, 237, 11, 169, 229, 149, 199, 78, 56, 15, 14, 190, 91, 216, 222, 247, 213, 74, 40, 8, 96, 20, 168, 119, 96, 26, 24, 52, 37, 82, 127, 57, 176, 147, 118, 59, 7, 224, 33, 117, 72, 155, 29, 82, 26, 215, 189, 140, 119, 28, 152, 118, 93, 222, 194, 192, 148, 115, 83, 253, 216, 212, 108, 88, 83, 175, 172, 220, 97, 79, 110, 42, 223, 170, 161, 34, 164, 144, 193, 76, 122, 92, 160, 41, 178, 175, 6, 35, 96, 113, 96, 158, 90, 129, 101, 26, 45, 70, 180, 189, 230, 15, 5, 247, 150, 209, 94, 171, 26, 13, 142, 212, 129, 1, 176, 5, 0, 112, 203, 174, 185, 119, 76, 233, 189, 54, 172, 189, 245, 223, 253, 205, 12, 88, 9, 126, 157, 225, 90, 40, 229, 191, 63, 30, 160, 224, 69, 3, 140, 109, 70, 89, 37, 213, 245, 194, 210, 180, 188, 63, 210, 139, 221, 2, 144, 200, 20, 177, 216, 29, 227, 242, 106, 12, 135, 142, 139, 144,

82, 225, 162, 171, 176, 108, 99, 6, 43, 193, 161, 116, 234, 216, 1, 242, 21, 124, 162, 98, 205, 124, 193, 38, 12, 242, 90, 101, 76, 204, 184, 124, 58, 180, 16, 240, 26, 76, 195, 250, 212, 191, 185, 191, 97, 198, 186, 73, 225, 75, 14, 90, 123, 121, 172, 101, 50, 160, 221, 141, 253, 205, 126, 77, 9, 87, 198, 110, 104, 182, 141, 120, 51, 25, 232, 3, 32, 80, 6, 156, 8, 18, 4, 135, 221, 142, 25, 135, 2, 129, 132, 115, 227, 74, 141, 28, 119, 11, 141, 117, 134, 198, 62, 150, 254, 97, 75, 197, 251, 99, 89, 204, 224, 226, 67, 83, 175, 89, 0, 81, 29, 38, 207, 89, 140, 255, 197, 177, 164, 128, 62, 116, 224, 180, 109, 169, 28, 2, 59, 176, 130, 252, 44, 178, 81, 24, 181, 176, 75, 44, 61, 91, 12, 37, 21, 255, 83, 130, 197, 16, 231, 60, 217, 56, 131, 118, 168, 202, 58, 52, 84, 124, 162, 185, 174, 162, 226, 242, 112, 68, 246, 202, 16, 208, 52, 154, 58, 129, 80, 102, 33, 171, 6, 186, 177, 14, 195, 88, 136, 6, 0, 155, 28, 100, 162, 207, 162, 222, 117, 248, 170, 208, 114, 87, 31, 57, 176, 33, 57, 83, 253, 12, 168, 110, 194, 59, 22, 86, 48, 227, 196, 22, 176, 218, 122, 149, 21, 249, 195, 178, 174, 250, 20, 34, 120, 60, 139, 201, 99, 40, 18, 177, 17, 54, 54, 6, 3, 222, 128, 160, 88, 11, 27, 0, 81, 192, 36, 41, 169, 146, 8, 47, 64, 136, 28, 64, 209, 67, 135, 202, 20, 234, 182, 91, 204, 146, 195, 187, 0, 72, 77, 11, 111, 152, 204, 252, 177, 212, 89, 33, 50, 132, 184, 44, 183, 186, 19, 250, 69, 176, 201, 102, 140, 14, 143, 212, 212, 160, 123, 208, 185, 27, 155, 68, 77, 133, 198, 2, 126, 155, 215, 22, 91, 30, 217, 176, 172, 244, 156, 174, 143, 75, 90, 21, 102, 1, 160, 59, 253, 188, 88, 57, 185, 197, 83, 24, 22, 180, 174, 47, 207, 52, 1, 141, 146, 119, 233, 68, 228, 224, 228, 193, 248, 155, 202, 90, 7, 213, 88, 33, 108, 107, 14, 86, 8, 120, 250, 58, 142, 35, 164, 238, 221, 219, 35, 123, 88, 199, 192, 143, 104, 83, 17, 166, 243, 247, 11, 166, 67, 68, 204, 132, 23, 110, 103, 228, 14, 55, 122, 88, 57, 180, 178, 237, 52, 130, 214, 245, 102, 123, 67, 73, 175, 1, 127, 112, 148, 94, 132, 164, 197, 153, 217, 87, 25, 89, 93, 63, 22, 66, 166, 90, 251, 101, 10, 145, 66, 17, 124, 36, 255, 165, 226, 97, 16, 86, 112, 154, 88, 105, 253, 56, 209, 229, 122, 103, 51, 24, 228, 190, 3, 236, 48, 182, 121, 176, 140, 128, 117, 87, 251, 224, 37, 23, 248, 21, 218, 85, 251, 136, 84, 147, 143, 144, 46, 155, 183, 251, 89, 86, 23, 26, 237, 100, 167, 32, 130, 173, 237, 89, 55, 110, 70, 142, 127, 65, 230, 208, 109, 69, 19, 253, 84, 130, 130, 193, 92, 58, 108, 150, 42, 136, 249, 234, 86, 241, 182, 19, 117, 246, 26, 181, 92, 101, 155, 44, 103, 235, 173, 30, 140, 90, 29, 183, 190, 77, 53, 206, 127, 5, 87, 8, 187, 184, 92, 4, 157, 22, 18, 105, 251, 39, 88, 182, 181, 103, 148, 233, 6, 63, 70, 188, 7, 101, 216, 127, 77, 31, 12, 233, 7, 147, 106, 30, 150, 77, 145, 13, 205, 48, 56, 245, 220, 89, 252, 127, 51, 180, 36, 31, 55, 18, 214, 230, 254, 217, 197, 65, 247, 27, 215, 117, 247, 108, 157, 121, 11, 63, 150, 195, 83, 6, 134, 242, 41, 24, 105, 204, 5, 63, 192, 14, 159, 113, 72, 140, 128, 51, 215, 80, 215, 39, 149, 94, 79, 128, 34, 5, 129, 82, 83, 121, 187, 37, 146, 27, 32, 177, 167, 71, 9, 195, 30, 199, 196, 205, 252, 207, 69, 8, 120, 27, 190, 51, 43, 75, 249, 234, 167, 116, 206, 203, 199, 43, 108, 87, 48, 155, 140, 228, 210, 85, 25, 161, 96, 67, 8, 205, 64, 39, 75, 88, 44, 238, 227, 16,

0, 100, 93, 129, 18, 4, 149, 50, 68, 72, 99, 35, 111, 254, 27, 102, 175, 108, 233, 87, 181, 44, 169, 18, 139, 79, 208, 14, 202, 192, 5, 162, 222, 231, 149, 24, 211, 49, 120, 101, 39, 206, 87, 147, 204, 200, 251, 104, 115, 5, 127, 117, 195, 79, 151, 18, 224, 52, 0, 245, 4, 85, 255, 103, 217, 0, 116, 198, 80, 91, 167, 192, 154, 199, 197, 149, 237, 51, 2, 131, 30, 226, 95, 105, 48, 68, 135, 208, 144, 120, 176, 145, 157, 8, 171, 80, 94, 61, 92, 92, 220, 157, 13, 138, 51, 23, 185, 124, 31, 77, 1, 87, 241, 43, 239, 55, 122, 86, 210, 48, 208, 204, 112, 144, 80, 147, 106, 219, 47, 253, 31, 134, 176, 16, 135, 219, 95, 17, 129, 83, 236, 125, 136, 112, 86, 228, 252, 71, 129, 218, 174, 156, 236, 12, 27, 159, 11, 138, 252, 253, 207, 31, 115, 214, 118, 239, 203, 16, 211, 205, 99, 22, 51, 163, 107, 162, 246, 199, 67, 127, 34, 108, 197, 53, 117, 58, 199, 3, 190, 74, 70, 190, 65, 235, 175, 97, 157, 215, 252, 189, 245, 100, 229, 248, 46, 90, 126, 237, 4, 159, 128, 58, 7, 156, 236, 69, 191, 85, 240, 179, 224, 249, 152, 49, 195, 223, 60, 78, 186, 157, 155, 217, 58, 105, 116, 164, 217, 111, 215, 150, 218, 252, 84, 86, 248, 140, 240, 226, 61, 106, 208, 95, 60, 163, 6, 0, 235, 253, 162, 96, 62, 234, 251, 249, 35, 21, 7, 211, 233, 86, 50, 33, 203, 67, 248, 60, 190, 123, 48, 167, 226, 90, 191, 71, 56, 183, 165, 17, 85, 76, 238, 140, 211, 168, 53, 223, 194, 4, 97, 149, 156, 120, 137, 76, 33, 229, 243, 194, 208, 198, 202, 139, 28, 114, 46, 224, 92, 254, 83, 100, 134, 158, 92, 70, 78, 61, 62, 138, 24, 173, 216, 66, 198, 70, 254, 47, 59, 193, 53, 6, 139, 19, 153, 253, 28, 199, 122, 160, 27, 67, 234, 209, 227, 139, 4, 50, 7, 178, 183, 89, 252, 32, 128, 137, 55, 52, 29, 89, 12, 111, 42, 181, 51, 170, 132, 132, 207, 170, 228, 254, 178, 213, 0, 136, 175, 8]

The resulting Authentication Tag value is:

[125, 249, 143, 191, 240, 4, 204, 132, 62, 241, 113, 178, 91, 88, 254, 19]

Encoding this JWE Ciphertext as BASE64URL(JWE Ciphertext) gives this value:

AwhB8lxlrlKjFn02LGWEqg27H4Tg9fyZAbFv3p5ZicHpj64QyHC44qqLZ3JEmnZTgQo
wIqZJ13jbyHB8LgePiqUJlhf6M2HPLgzW8L-mEeQ0jvDUTrE07NtOerBk8bwBQyZ6g
0kQ3DEOIglfYxV8-FJvNBYwbqN1Bck6d_i70tjSHV-8DIrp-3JcRIe05YKy30i34Z_
GOiAc1EK21B1lc_AE11PII_wvvtRiUiG8YofQXakWd1_098Kap-UgmyWPfreUJ3lJP
nbD4Ve95owEfMGLOPflo2MnjaTDCwQokoJ_xplQ2vNPz8iguLcHBoKllyQFJL2mOWB
wqhBo9Oj-0800as5mmLsvQMTflIrIEbbTMzHMBZ8EFW9fWwwFu0DWQJGkMNhmBZQ-3
lvqTc-M6-gWA6D8PDhONfP2Oib2HGizwG1iEaX8GRyUpfLuljCLiElDkGOewhKuKkZ
h04DKNM5Nbugf2atmU9OP0Ldx5peCUtRGlgMVl7Qup5ZXHTjgPDr5b2N731UooCGAU
qHdgGhg0JVJ_ObCTdjsH4CF1SJsduhrXvYx3HJh2Xd7CwJRzU_3Y1GxYU6-s3GFPbi
rfqqEipJDBTHpcoCmyrWYjYHFgnlqBZRotRrS95g8F95bRXqsaDY7UgQGwBQBwy665
d0zpvTasvxf_c0MWA1-neFaKOW_Px6g4EUDjG1GWSXV9cLStLw_0ovdApDIFLHYHe
PyagyHjouQUuGiq7BsYwYrwaF06tgB8hV8omLNfMEMDPJaZUzMuHw6tBDWgkzD-ts_
ub9hxrj4UsOWnt5rGUyoN2N_c1-TQlXxm5oto14MxnoAyBQBpwIEgSH3Y4ZhwKBhH
PjSo0cdwuNdYbGPPb-YUvF-2NZzODiQ1OvWQBRHSbPWYz_xbGkgD504LrtqRwCO7CC
_CyyURilsEssPVsMJRX_U4LFEoc82TiDdqjK0jRUfKK5rqLi8nBE9soQ0DSaOoFQZi
GrBrqDsNYiAYAmxxkos-i3nX4qtByVx85sCE5U_0MqG7COxZWMOPeFrDaepUV-cOy
rvoUIng8i8ljkBKxETY2BgPegKBYCxsAUcAkKamSCC9AiBxA0UOHYhTqtlvMks07AE
hNC2-YzPyx1FkhMoS4LLe6E_pFsMlmjA6PlNSge9C5G5tETyXGAN6blxZbHtmwrPSc
ro9LWhVmAA7_bxYObnFUxgWtK4vzzQBjZJ36UTk4OTB-JvKWgfvVWCFsaw5WCHj6Oo
4jp07d2yN7WMfAj2hTEabz9wumQ0TMhBduZ-QON3pYObSy7TSClVvme0NJrWf_cJRe
hKTFmdlXGvldPxZCplR7ZQqRQhF8JP-14mEQVnCaWgn9ONHlcmczGOS-A-wwtnmwjI
BlV_vgJRf4FdpV-4hUk4-QLpu3-1lWFxrtZKcggq3tWTduRo5_QebQbUUT_VSCgsFc
OmyWkoj56lxbxthN19hq1XGWbLGfrrR6MWh23vk01zn8FVwi7uFwEnRYSafsnWLa1Z5
TpBj9GvAdl2H9NHwzpB5NqHpZNkQ3NMDj13Fn8fz00JB83Etbm_tnFQfcb13X3bJ15
Cz-Ww1MGhvIpGGnMBT_ADp9xSIyAM9dQlyeVXk-AIGWBULN5uyWSGyCxp0cJwx7HxM
38z0UIeBu-MytL-eqndM7LxytsVzCbJOTSVRmhYEMiZUAnSlgs7uMQAGRdgrIElTJE
SGMjb_4bZq9s6Ve1LKkSi0_QDsrABaLe55UY0zF4ZSfOV5PMYPtoCwV_dcNPlxLgNA
D1BFX_Z9kAdMZQW6fAmsfFle0zAoMe4l9pMESH0JB4sJGdCKtQXj1cXNydyYozF718
H00BV_Er7zd6VtIw0MxwkFCTatsv_R-GsBCH218RgVPsfYhwVuT8R4HarpzSDBufC4
r8_c8fc9Z278sQ081jFjOja6L2x0N_ImzFNXU6xwO-Ska-QeuvYZ3X_L31ZOX4Llp-
7QSfgDoHnOxFv1Xws-D5mDHD3zxOup2b2TppdKTZb9eW2vxUVviM8OI9atBfPKMGAO
v9omA-6vv5IxUH0-lWmiHLQ_g8vnswp-Jav0c4t6URVUzuJN0oNd_CBGGVnHiJTCH1
88LQxsqLHHIu4Fz-U2SGnlxGTj0-ihit2ELGRv4v08E1BosTmf0cx3qgG0Pq0eOLBD
IHsrdZ_CCAiTc0HVkMbyq1M6qEhM-q5P6y1QCirwg

Encoding this JWE Authentication Tag as BASE64URL(JWE Authentication Tag) gives this value:

ffmPv_AEzIQ-8XGyWlj-Ew

C.8. Complete Representation

Assemble the final representation: The Compact Serialization of this result is the string BASE64URL(UTF8(JWE Protected Header)) || '.' || BASE64URL(JWE Encrypted Key) || '.' || BASE64URL(JWE Initialization Vector) || '.' || BASE64URL(JWE Ciphertext) || '.' || BASE64URL(JWE Authentication Tag).

The final result in this example is:

```
eyJhbGciOiJQQkVMTi1IUzI1NitBMTI4S1ciLCJwMnMiOiIyV0NUY0paMVJ2ZF9DSn
VKcmlwUTF3IiwicDJjIjo0MDk2LCJlbmMiOiJBMTI4Q0JDLUhTMjU2IiwiY3R5Ijo1
andrK2pzb24ifQ.
yeyPcAzqyNMh8f9BcD-skmlregAeFSwVDj3IOR795FPaUopQef7BeQ.
Ye9j1qs22DmRSAddIh-VnA.
AwhB81xrlKjFn02LGWEqg27H4Tg9fyZAbFv3p5ZicHpj64QyHC44qqL3ZJEmnZTgQo
wIqZJ13jbyHB8LgePiqUJ1hf6M2HPLgzW8L-mEeQ0jvDUTrE07NtOerBk8bwBQyZ6g
0kQ3DEOIglfYxV8-FJvNBYwbqN1Bck6d_i7OtjSHV-8DIrp-3JcRIe05YKy3Oi34Z_
GOiAc1EK21B1lc_AE11PII_wvvtRiUiG8YofQXakWd1_098Kap-UgmyWPfreUJ3lJP
nbd4Ve95owEfMGLOPflo2MnjaTDCwQokoJ_xplQ2vNPz8iguLchBoKllyQFJL2mOWB
wqhBo9Oj-0800as5mmLsvQMTflIrIEbbTMzHMBZ8EFW9fWwwFu0DWQJGkMNhmBZQ-3
lvqTc-M6-gWA6D8PDhONfP2Oib2HGizwG1iEaX8GRyUpfLuljCLiElDkGOewhKuKkZ
h04DKNM5Nbugf2atmU9OP0Ldx5peCUtRG1gMVl7Qup5ZXHTjgPDR5b2N731UooCGAU
qHdgGhg0JVJ_ObCTdjsH4CF1SJsdUhrXvYx3HJh2Xd7CwJRzU_3Y1GxYU6-s3GFPbi
rfqqEipJDBTHpcoCmyrwyjYHFgnlqBZRotRrS95g8F95bRXqsaDY7UgQGwBQbwy665
d0zpvTasvfXf_c0MWA1-neFaKOW_Px6g4EUDjG1GWSXV9cLStLw_0ovdApDIFLHYHe
PyagyHjouQUuGiq7BsYwYrwaF06tgB8hV8omLNfMEMDPJaZUzMuHw6tBDWgkzD-ts_
ub9hxrPJ4UsOWnt5rGUyoN2N_c1-TQlXxm5oto14MxnoAyBQBpwIEgSH3Y4ZhWKBhH
PjSo0cdwuNdYbGppb-YUvF-2NZzODiQ1OvWQBRHSbPWYz_xbGkgD504LrtqRwCO7CC
_CyyURilsEssPVsMJRX_U4LFEoc82TiDdqjKOjRufKK5rqLi8nBE9soQ0DSaOoFQZi
GrBrqxDSNYiAYAmxxkos-i3nX4qtByVx85sCE5U_0MqG7CoxZWMOPeFrDaepUV-cOy
rvoUing8i8ljKBKxETY2BgPegKBYCxsAUcAkKamSCC9AiBxA0UOHytTqtlvMks07AE
hNC2-YzPyx1FkhMoS4LLe6E_pFsMlmjA6PlNSge9C5G5tETyXGAn6blxZbHtmwrPSc
ro9LWhVmAA7_bxYObnFUxgWtK4vzzQBjZJ36UTk4OTB-JvKWgfVWCFsaw5WCHj6Oo
4jp07d2yN7WMfAj2hTEabz9wumQ0TMhBduZ-QON3pYObSy7TSC1vVme0NJrWf_cJRe
hKTFmdlXGVldPxZCplR7ZQqRQhF8JP-14mEQVnCaWgn9ONHlemczGOS-A-wwtnmwjI
BlV_vgJRf4FdpV-4hUk4-QLpu3-1lWFxrtZKcggq3tWTduRo5_QebQbUUT_VSCgsFc
OmyWKOj56lxbxthN19hq1XGWbLGfrrR6MWh23vk01zn8FVwi7uFwEnRYSafsnWLa1Z5
TpBj9GvAdl2H9NHwzpB5NqHpZNkQ3NMDj13Fn8fz00JB83Etbm_tnFQfcb13X3bJ15
Cz-Ww1MGhvIpGGnMBT_AdP9xSIyAM9dQ1yeVXk-AIGWBu1N5uyWSGyCxp0cJwx7HxM
38z0UIeBu-MytL-eqndM7LxytsVzCbJOTSVRmHYEMIzUAnSlgs7uMQAGRdgRIElTJE
SGMjb_4bzq9s6Ve1LKkSi0_QDsRABaLe55UY0zF4ZSfOV5PMYPtocyV_dcNPlxLgNA
DlBFX_Z9kAdMZQW6fAmsfFle0zAoMe4l9pMESH0JB4sJGdCKtQXj1cXNydyYozF7l8
H00BV_Er7zd6VtIw0MxwkFCTatsv_R-GsBCH218RgVPsfYhwVuT8R4HarpsZDBufC4
r8_c8fc9Z278sQ081jFjOja6L2x0N_ImzFNXU6xw0-Ska-QeuvYZ3X_L31ZOX4Llp-
7QSfgDoHnOxFv1Xws-D5mDHD3zxOup2b2TppdKTZb9eW2vxUVviM8OI9atBfPKMGAO
v9omA-6vv5IxUH0-lWmiHLQ_g8vnswp-Jav0c4t6URVUzujNOoNd_CBGgVnHiJTCH1
88LQxsqLHHIu4Fz-U2SGnlxGTj0-ihit2ELGRv4v08E1BosTmf0cx3qgG0Pq0eOLBD
IHsrdZ_CCAiTc0HVkMbyq1M6qEhM-q5P6ylQCirwg.
ffmPv_AEzIQ-8XGyW1j-Ew
```

Appendix D. Acknowledgements

A JSON representation for RSA public keys was previously introduced by John Panzer, Ben Laurie, and Dirk Balfanz in Magic Signatures

[MagicSignatures].

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, Joe Hildebrand, Edmund Jay, Ben Laurie, James Manger, Matt Miller, Tony Nadalin, Axel Nennker, John Panzer, Eric Rescorla, Nat Sakimura, Jim Schaad, Paul Tarjan, 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]]

-17

- o Refined the "typ" and "cty" definitions to always be MIME Media Types, with the omission of "application/" prefixes recommended for brevity, addressing issue #50.
- o Added an example encrypting an RSA private key with "PBES2-HS256+A128KW" and "A128CBC-HS256". Thanks to Matt Miller for producing this!
- o Processing rules occurring in both JWS and JWK are now referenced in JWS by JWK, rather than duplicated, addressing issue #57.
- o 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.

-16

- o Changes to address editorial and minor issues #41, #42, #43, #47, #51, #67, #71, #76, #80, #83, #84, #85, #86, #87, and #88.

-15

- o Changes to address editorial issues #48, #64, #65, #66, and #91.

-14

- o Relaxed language introducing key parameters since some parameters are applicable to multiple, but not all, key types.

-13

- o Applied spelling and grammar corrections.

-12

- o Stated that recipients MUST either reject JWKs and JWK Sets with duplicate member names or use a JSON parser that returns only the lexically last duplicate member name.

-11

- o Stated that when "kid" values are used within a JWK Set, different keys within the JWK Set SHOULD use distinct "kid" values.
- o Added optional "x5u" (X.509 URL), "x5t" (X.509 Certificate Thumbprint), and "x5c" (X.509 Certificate Chain) JWK parameters.
- o Added section on Encrypted JWK and Encrypted JWK Set Formats.
- o Added a Parameter Information Class value to the JSON Web Key Parameters registry, which registers whether the parameter conveys public or private information.
- o Registered "application/jwk+json" and "application/jwk-set+json" MIME types and "JWK" and "JWK-SET" typ header parameter values, addressing issue #21.

-10

- o No changes were made, other than to the version number and date.

-09

- o Expanded the scope of the JWK specification to include private and symmetric key representations, as specified by draft-jones-jose-json-private-and-symmetric-key-00.
- o Defined that members that are not understood must be ignored.

-08

- o Changed the name of the JWK key type parameter from "alg" to "kty" to enable use of "alg" to indicate the particular algorithm that the key is intended to be used with.
- o Clarified statements of the form "This member is OPTIONAL" to "Use of this member is OPTIONAL".
- o Referenced String Comparison Rules in JWS.
- o Added seriesInfo information to Internet Draft references.

-07

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

-06

- o Changed the name of the JWK RSA exponent parameter from "exp" to "xpo" so as to allow the potential use of the name "exp" for a future extension that might define an expiration parameter for keys. (The "exp" name is already used for this purpose in the JWT specification.)
- o Clarify that the "alg" (algorithm family) member is REQUIRED.
- o Correct an instance of "JWK" that should have been "JWK Set".
- o Applied changes made by the RFC Editor to RFC 6749's registry language to this specification.

-05

- o Indented artwork elements to better distinguish them from the body text.

-04

- o Refer to the registries as the primary sources of defined values and then secondarily reference the sections defining the initial contents of the registries.
- o Normatively reference XML DSIG 2.0 [W3C.CR-xmlsig-core2-20120124] for its security considerations.

- o Added this language to Registration Templates: "This name is case sensitive. Names that match other registered names in a case insensitive manner SHOULD NOT be accepted."
- o Described additional open issues.
- o Applied editorial suggestions.

-03

- o Clarified that "kid" values need not be unique within a JWK Set.
- o Moved JSON Web Key Parameters registry to the JWK specification.
- o Added "Collision Resistant Namespace" to the terminology section.
- o Changed registration requirements from RFC Required to Specification Required with Expert Review.
- o Added Registration Template sections for defined registries.
- o Added Registry Contents sections to populate registry values.
- o Numerous editorial improvements.

-02

- o Simplified JWK terminology to get replace the "JWK Key Object" and "JWK Container Object" terms with simply "JSON Web Key (JWK)" and "JSON Web Key Set (JWK Set)" and to eliminate potential confusion between single keys and sets of keys. As part of this change, the top-level member name for a set of keys was changed from "jwk" to "keys".
- o Clarified that values with duplicate member names MUST be rejected.
- o Established JSON Web Key Set Parameters registry.
- o Explicitly listed non-goals in the introduction.
- o Moved algorithm-specific definitions from JWK to JWA.
- o Reformatted to give each member definition its own section heading.

-01

- o Corrected the Magic Signatures reference.

-00

- o Created the initial IETF draft based upon draft-jones-json-web-key-03 with no normative changes.

Author's Address

Michael B. Jones
Microsoft

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

JOSE Working Group
Internet-Draft
Intended status: Standards Track
Expires: April 10, 2014

M. Jones
Microsoft
J. Bradley
Ping Identity
N. Sakimura
NRI
October 7, 2013

JSON Web Signature (JWS)
draft-ietf-jose-json-web-signature-17

Abstract

JSON Web Signature (JWS) represents content secured with digital signatures or Message Authentication Codes (MACs) using JavaScript Object Notation (JSON) based data structures. Cryptographic algorithms and identifiers for use with this specification are described in the separate JSON Web Algorithms (JWA) specification and an IANA registry defined by that specification. Related encryption capabilities are described in the separate JSON Web Encryption (JWE) specification.

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 April 10, 2014.

Copyright Notice

Copyright (c) 2013 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<http://trustee.ietf.org/license-info>) in effect on the date of

publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction	4
1.1. Notational Conventions	4
2. Terminology	5
3. JSON Web Signature (JWS) Overview	6
3.1. Example JWS	7
4. JWS Header	9
4.1. Registered Header Parameter Names	9
4.1.1. "alg" (Algorithm) Header Parameter	9
4.1.2. "jku" (JWK Set URL) Header Parameter	10
4.1.3. "jwk" (JSON Web Key) Header Parameter	10
4.1.4. "x5u" (X.509 URL) Header Parameter	10
4.1.5. "x5t" (X.509 Certificate SHA-1 Thumbprint) Header Parameter	11
4.1.6. "x5c" (X.509 Certificate Chain) Header Parameter	11
4.1.7. "kid" (Key ID) Header Parameter	11
4.1.8. "typ" (Type) Header Parameter	12
4.1.9. "cty" (Content Type) Header Parameter	12
4.1.10. "crit" (Critical) Header Parameter	13
4.2. Public Header Parameter Names	13
4.3. Private Header Parameter Names	13
5. Producing and Consuming JWSS	14
5.1. Message Signing or MACing	14
5.2. Message Signature or MAC Validation	14
5.3. String Comparison Rules	16
6. Key Identification	16
7. Serializations	17
7.1. JWS Compact Serialization	17
7.2. JWS JSON Serialization	17
8. IANA Considerations	19
8.1. JSON Web Signature and Encryption Header Parameters Registry	20
8.1.1. Registration Template	20
8.1.2. Initial Registry Contents	20
8.2. Media Type Registration	22
8.2.1. Registry Contents	22
9. Security Considerations	23
9.1. Cryptographic Security Considerations	23
9.2. JSON Security Considerations	24

9.3. Unicode Comparison Security Considerations	25
9.4. TLS Requirements	25
10. References	25
10.1. Normative References	25
10.2. Informative References	27
Appendix A. JWS Examples	28
A.1. Example JWS using HMAC SHA-256	28
A.1.1. Encoding	28
A.1.2. Validating	30
A.2. Example JWS using RSASSA-PKCS-v1_5 SHA-256	30
A.2.1. Encoding	30
A.2.2. Validating	33
A.3. Example JWS using ECDSA P-256 SHA-256	33
A.3.1. Encoding	33
A.3.2. Validating	35
A.4. Example JWS using ECDSA P-521 SHA-512	35
A.4.1. Encoding	36
A.4.2. Validating	38
A.5. Example Plaintext JWS	38
A.6. Example JWS Using JWS JSON Serialization	39
A.6.1. JWS Per-Signature Protected Headers	39
A.6.2. JWS Per-Signature Unprotected Headers	40
A.6.3. Complete JWS Header Values	40
A.6.4. Complete JWS JSON Serialization Representation	40
Appendix B. "x5c" (X.509 Certificate Chain) Example	41
Appendix C. Notes on implementing base64url encoding without padding	43
Appendix D. Negative Test Case for "crit" Header Parameter	44
Appendix E. Acknowledgements	44
Appendix F. Document History	45
Authors' Addresses	51

1. Introduction

JSON Web Signature (JWS) represents content secured with digital signatures or Message Authentication Codes (MACs) using JavaScript Object Notation (JSON) [RFC4627] based data structures. The JWS cryptographic mechanisms provide integrity protection for an arbitrary sequence of octets.

Two closely related serializations for JWS objects are defined. The JWS Compact Serialization is a compact, URL-safe representation intended for space constrained environments such as HTTP Authorization headers and URI query parameters. The JWS JSON Serialization represents JWS objects as JSON objects and enables multiple signatures and/or MACs to be applied to the same content. Both share the same cryptographic underpinnings.

Cryptographic algorithms and identifiers for use with this specification are described in the separate JSON Web Algorithms (JWA) [JWA] specification and an IANA registry defined by that specification. Related encryption capabilities are described in the separate JSON Web Encryption (JWE) [JWE] specification.

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

JWS Web Signature (JWS) A data structure representing a digitally signed or MACed message.

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

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

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

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

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

Header Parameter A name/value pair that is member of the JWS Header.

Base64url Encoding Base64 encoding using the URL- and filename-safe character set defined in Section 5 of RFC 4648 [RFC4648], with all trailing '=' characters omitted (as permitted by Section 3.2). (See Appendix C for notes on implementing base64url encoding without padding.)

JWS Signing Input The input to the digital signature or MAC computation. Its value is ASCII(BASE64URL(UTF8(JWS Protected Header)) || '.' || BASE64URL(JWS Payload)).

JWS Compact Serialization A representation of the JWS as the string BASE64URL(UTF8(JWS Protected Header)) || '.' || BASE64URL(JWS Payload) || '.' || BASE64URL(JWS Signature). This representation is compact and URL-safe.

JWS JSON Serialization A representation of the JWS as a JSON structure. Unlike the JWS Compact Serialization, the JWS JSON Serialization enables multiple digital signatures and/or MACs to be applied to the same content. This representation is neither compact nor URL-safe.

Collision Resistant Name A name in a namespace that enables names to be allocated in a manner such that they are highly unlikely to collide with other names. Examples of collision resistant namespaces include: Domain Names, Object Identifiers (OIDs) as defined in the ITU-T X.660 and X.670 Recommendation series, and Universally Unique Identifiers (UUIDs) [RFC4122]. When using an administratively delegated namespace, the definer of a name needs to take reasonable precautions to ensure they are in control of the portion of the namespace they use to define the name.

StringOrURI A JSON string value, with the additional requirement that while arbitrary string values MAY be used, any value containing a ":" character MUST be a URI [RFC3986]. StringOrURI values are compared as case-sensitive strings with no transformations or canonicalizations applied.

3. JSON Web Signature (JWS) Overview

JWS represents digitally signed or MACed content using JSON data structures and base64url encoding. A JWS represents these logical values:

JWS Header JSON object containing the parameters describing the cryptographic operations and parameters employed. The JWE Header members are the union of the members of the JWS Protected Header and the JWS Unprotected Header, as described below.

JWS Payload Message content to be secured.

JWS Signature Digital signature or MAC over the JWS Protected Header and the JWS Payload.

The JWS Header represents the combination of these logical values:

JWS Protected Header JSON object containing some of the parameters describing the cryptographic operations and parameters employed. This value is integrity protected in the digital signature or MAC calculation of the JWS Signature.

JWS Unprotected Header JSON object containing some of the parameters describing the cryptographic operations and parameters employed. This value is not integrity protected in the digital signature or MAC calculation of the JWS Signature.

This document defines two serializations for JWS objects: a compact, URL-safe serialization called the JWS Compact Serialization and a JSON serialization called the JWS JSON Serialization. In both serializations, the JWS Protected Header, JWS Payload, and JWS Signature are base64url encoded for transmission, since JSON lacks a way to directly represent octet sequences.

In the JWS Compact Serialization, no JWS Unprotected Header is used. In this case, the JWS Header and the JWS Protected Header are the same.

In the JWS Compact Serialization, a JWS object is represented as the combination of these three string values,
BASE64URL(UTF8(JWS Protected Header)),
BASE64URL(JWS Payload), and
BASE64URL(JWS Signature),
concatenated in that order, with the three strings being separated by two period ('.') characters.

In the JWS JSON Serialization, one or both of the JWS Protected Header and JWS Unprotected Header MUST be present. In this case, the members of the JWS Header are the combination of the members of the JWS Protected Header and the JWS Unprotected Header values that are present.

In the JWS JSON Serialization, a JWS object is represented as the combination of these four values,
BASE64URL(UTF8(JWS Protected Header)),
JWS Unprotected Header,
BASE64URL(JWS Payload), and
BASE64URL(JWS Signature),
with the three base64url encoding result strings and the JWS Unprotected Header value being represented as members within a JSON object. The inclusion of some of these values is OPTIONAL. The JWS JSON Serialization can also represent multiple signature and/or MAC values, rather than just one. See Section 7.2 for more information about the JWS JSON Serialization.

3.1. Example JWS

This section provides an example of a JWS. Its computation is described in more detail in Appendix A.1, including specifying the exact octet sequences representing the JSON values used and the key

value used.

The following example JWS Protected Header declares that the encoded object is a JSON Web Token (JWT) [JWT] and the JWS Protected Header and the JWS Payload are secured using the HMAC SHA-256 algorithm:

```
{ "typ": "JWT",  
  "alg": "HS256" }
```

Encoding this JWS Protected Header as `BASE64URL(UTF8(JWS Protected Header))` gives this value:

```
eyJ0eXAiOiJKV1QiLA0KICJhbGciOiJIUzI1NiJ9
```

The UTF-8 representation of following JSON object is used as the JWS Payload. (Note that the payload can be any content, and need not be a representation of a JSON object.)

```
{ "iss": "joe",  
  "exp": 1300819380,  
  "http://example.com/is_root": true }
```

Encoding this JWS Payload as `BASE64URL(JWS Payload)` gives this value (with line breaks for display purposes only):

```
eyJpc3MiOiJqb2UiLA0KICJleHAiOjEzMDA4MTkzODAsDQogImh0dHA6Ly9leGft  
cGxlLmNvbS9pc19yb290Ijp0cnVlfQ
```

Computing the HMAC of the JWS Signing Input `ASCII(BASE64URL(UTF8(JWS Protected Header)) || '.' || BASE64URL(JWS Payload))` with the HMAC SHA-256 algorithm using the key specified in Appendix A.1 and `base64url` encoding the result yields this `BASE64URL(JWS Signature)` value:

```
dBjftJeZ4CVP-mB92K27uhbUJU1plr_wW1lgFWFOEjXk
```

Concatenating these values in the order `Header.Payload.Signature` with period ('.') characters between the parts yields this complete JWS representation using the JWS Compact Serialization (with line breaks for display purposes only):

```
eyJ0eXAiOiJKV1QiLA0KICJhbGciOiJIUzI1NiJ9  
.  
eyJpc3MiOiJqb2UiLA0KICJleHAiOjEzMDA4MTkzODAsDQogImh0dHA6Ly9leGft  
cGxlLmNvbS9pc19yb290Ijp0cnVlfQ  
.  
dBjftJeZ4CVP-mB92K27uhbUJU1plr_wW1lgFWFOEjXk
```

See Appendix A for additional examples.

4. JWS Header

The members of the JSON object(s) representing the JWS Header describe the digital signature or MAC applied to the JWS Protected Header and the JWS Payload and optionally additional properties of the JWS. The Header Parameter names within the JWS Header MUST be unique; recipients MUST either reject JWSs with duplicate Header Parameter names or use a JSON parser that returns only the lexically last duplicate member name, as specified in Section 15.12 (The JSON Object) of ECMA Script 5.1 [ECMA Script].

Implementations are required to understand the specific Header Parameters defined by this specification that are designated as "MUST be understood" and process them in the manner defined in this specification. All other Header Parameters defined by this specification that are not so designated MUST be ignored when not understood. Unless listed as a critical Header Parameter, per Section 4.1.10, all Header Parameters not defined by this specification MUST be ignored when not understood.

There are three classes of Header Parameter names: Registered Header Parameter names, Public Header Parameter names, and Private Header Parameter names.

4.1. Registered Header Parameter Names

The following Header Parameter names are registered in the IANA JSON Web Signature and Encryption Header Parameters registry defined in Section 8.1, with meanings as defined below.

As indicated by the common registry, JWSs and JWEs share a common Header Parameter space; when a parameter is used by both specifications, its usage must be compatible between the specifications.

4.1.1. "alg" (Algorithm) Header Parameter

The "alg" (algorithm) Header Parameter identifies the cryptographic algorithm used to secure the JWS. The signature, MAC, or plaintext value is not valid if the "alg" value does not represent a supported algorithm, or if there is not a key for use with that algorithm associated with the party that digitally signed or MACed the content. "alg" values SHOULD either be registered in the IANA JSON Web Signature and Encryption Algorithms registry defined in [JWA] or be a value that contains a Collision Resistant Name. The "alg" value is a

case sensitive string containing a StringOrURI value. Use of this Header Parameter is REQUIRED. This Header Parameter MUST be understood and processed by implementations.

A list of defined "alg" values for this use can be found in the IANA JSON Web Signature and Encryption Algorithms registry defined in [JWA]; the initial contents of this registry are the values defined in Section 3.1 of the JSON Web Algorithms (JWA) [JWA] specification.

4.1.2. "jku" (JWK Set URL) Header Parameter

The "jku" (JWK Set URL) Header Parameter is a URI [RFC3986] that refers to a resource for a set of JSON-encoded public keys, one of which corresponds to the key used to digitally sign the JWS. The keys MUST be encoded as a JSON Web Key Set (JWK Set) [JWK]. The protocol used to acquire the resource MUST provide integrity protection; an HTTP GET request to retrieve the JWK Set MUST use TLS [RFC2818] [RFC5246]; the identity of the server MUST be validated, as per Section 3.1 of HTTP Over TLS [RFC2818]. Use of this Header Parameter is OPTIONAL.

4.1.3. "jwk" (JSON Web Key) Header Parameter

The "jwk" (JSON Web Key) Header Parameter is the public key that corresponds to the key used to digitally sign the JWS. This key is represented as a JSON Web Key [JWK]. Use of this Header Parameter is OPTIONAL.

4.1.4. "x5u" (X.509 URL) Header Parameter

The "x5u" (X.509 URL) Header Parameter is a URI [RFC3986] that refers to a resource for the X.509 public key certificate or certificate chain [RFC5280] corresponding to the key used to digitally sign the JWS. The identified resource MUST provide a representation of the certificate or certificate chain that conforms to RFC 5280 [RFC5280] in PEM encoded form [RFC1421]. The certificate containing the public key corresponding to the key used to digitally sign the JWS MUST be the first certificate. This MAY be followed by additional certificates, with each subsequent certificate being the one used to certify the previous one. The protocol used to acquire the resource MUST provide integrity protection; an HTTP GET request to retrieve the certificate MUST use TLS [RFC2818] [RFC5246]; the identity of the server MUST be validated, as per Section 3.1 of HTTP Over TLS [RFC2818]. Use of this Header Parameter is OPTIONAL.

4.1.5. "x5t" (X.509 Certificate SHA-1 Thumbprint) Header Parameter

The "x5t" (X.509 Certificate SHA-1 Thumbprint) Header Parameter is a base64url encoded SHA-1 thumbprint (a.k.a. digest) of the DER encoding of the X.509 certificate [RFC5280] corresponding to the key used to digitally sign the JWS. Use of this Header Parameter is OPTIONAL.

If, in the future, certificate thumbprints need to be computed using hash functions other than SHA-1, it is suggested that additional related Header Parameters be defined for that purpose. For example, it is suggested that a new "x5t#S256" (X.509 Certificate Thumbprint using SHA-256) Header Parameter could be defined by registering it in the IANA JSON Web Signature and Encryption Header Parameters registry defined in Section 8.1.

4.1.6. "x5c" (X.509 Certificate Chain) Header Parameter

The "x5c" (X.509 Certificate Chain) Header Parameter contains the X.509 public key certificate or certificate chain [RFC5280] corresponding to the key used to digitally sign the JWS. The certificate or certificate chain is represented as a JSON array of certificate value strings. Each string in the array is a base64 encoded ([RFC4648] Section 4 -- not base64url encoded) DER [ITU.X690.1994] PKIX certificate value. The certificate containing the public key corresponding to the key used to digitally sign the JWS MUST be the first certificate. This MAY be followed by additional certificates, with each subsequent certificate being the one used to certify the previous one. The recipient MUST verify the certificate chain according to [RFC5280] and reject the signature if any validation failure occurs. Use of this Header Parameter is OPTIONAL.

See Appendix B for an example "x5c" value.

4.1.7. "kid" (Key ID) Header Parameter

The "kid" (key ID) Header Parameter is a hint indicating which key was used to secure the JWS. This parameter allows originators to explicitly signal a change of key to recipients. Should the recipient be unable to locate a key corresponding to the "kid" value, they SHOULD treat that condition as an error. The interpretation of the "kid" value is unspecified. Its value MUST be a string. Use of this Header Parameter is OPTIONAL.

When used with a JWK, the "kid" value can be used to match a JWK "kid" parameter value.

4.1.1.8. "typ" (Type) Header Parameter

The "typ" (type) Header Parameter is used to declare the MIME Media Type [IANA.MediaTypes] of this complete JWS object in contexts where this is useful to the application. This parameter has no effect upon the JWS processing. Use of this Header Parameter is OPTIONAL.

Per [RFC2045], all media type values, subtype values, and parameter names are case-insensitive. However, parameter values are case-sensitive unless otherwise specified for the specific parameter.

To keep messages compact in common situations, it is RECOMMENDED that senders omit an "application/" prefix of a media type value in a "typ" Header Parameter when no other '/' appears in the media type value. A recipient using the media type value MUST treat it as if "application/" were prepended to any "typ" value not containing a '/'. For instance, a "typ" value of "example" SHOULD be used to represent the "application/example" media type.

The "typ" value "JOSE" can be used by applications to indicate that this object is a JWS or JWE using the JWS Compact Serialization or the JWE Compact Serialization. The "typ" value "JOSE+JSON" can be used by applications to indicate that this object is a JWS or JWE using the JWS JSON Serialization or the JWE JSON Serialization. Other type values can also be used by applications.

4.1.1.9. "cty" (Content Type) Header Parameter

The "cty" (content type) Header Parameter is used to declare the MIME Media Type [IANA.MediaTypes] of the secured content (the payload) in contexts where this is useful to the application. This parameter has no effect upon the JWS processing. Use of this Header Parameter is OPTIONAL.

Per [RFC2045], all media type values, subtype values, and parameter names are case-insensitive. However, parameter values are case-sensitive unless otherwise specified for the specific parameter.

To keep messages compact in common situations, it is RECOMMENDED that senders omit an "application/" prefix of a media type value in a "cty" Header Parameter when no other '/' appears in the media type value. A recipient using the media type value MUST treat it as if "application/" were prepended to any "cty" value not containing a '/'. For instance, a "cty" value of "example" SHOULD be used to represent the "application/example" media type.

4.1.10. "crit" (Critical) Header Parameter

The "crit" (critical) Header Parameter indicates that extensions to [[this specification]] are being used that MUST be understood and processed. Its value is an array listing the Header Parameter names defined by those extensions that are used in the JWS Header. If any of the listed extension Header Parameters are not understood and supported by the receiver, it MUST reject the JWS. Senders MUST NOT include Header Parameter names defined by [[this specification]] or by [JWA] for use with JWS, duplicate names, or names that do not occur as Header Parameter names within the JWS Header in the "crit" list. Senders MUST not use the empty list "[]" as the "crit" value. Recipients MAY reject the JWS if the critical list contains any Header Parameter names defined by [[this specification]] or by [JWA] for use with JWS, or any other constraints on its use are violated. This Header Parameter MUST be integrity protected, and therefore MUST occur only with the JWS Protected Header, when used. Use of this Header Parameter is OPTIONAL. This Header Parameter MUST be understood and processed by implementations.

An example use, along with a hypothetical "exp" (expiration-time) field is:

```
{ "alg": "ES256",  
  "crit": [ "exp" ],  
  "exp": 1363284000  
}
```

4.2. Public Header Parameter Names

Additional Header Parameter names can be defined by those using JWSs. However, in order to prevent collisions, any new Header Parameter name SHOULD either be registered in the IANA JSON Web Signature and Encryption Header Parameters registry defined in Section 8.1 or be a Public Name: a value that contains a Collision Resistant Name. In each case, the definer of the name or value needs to take reasonable precautions to make sure they are in control of the part of the namespace they use to define the Header Parameter name.

New Header Parameters should be introduced sparingly, as they can result in non-interoperable JWSs.

4.3. Private Header Parameter Names

A producer and consumer of a JWS may agree to use Header Parameter names that are Private Names: names that are not Registered Header Parameter names Section 4.1 or Public Header Parameter names Section 4.2. Unlike Public Header Parameter names, Private Header

Parameter names are subject to collision and should be used with caution.

5. Producing and Consuming JWSs

5.1. Message Signing or MACing

To create a JWS, one MUST perform these steps. The order of the steps is not significant in cases where there are no dependencies between the inputs and outputs of the steps.

1. Create the content to be used as the JWS Payload.
2. Compute the encoded payload value `BASE64URL(JWS Payload)`.
3. Create a JWS Header containing the desired set of Header Parameters. Note that white space is explicitly allowed in the representation and no canonicalization need be performed before encoding.
4. Compute the encoded header value `BASE64URL(UTF8(JWS Protected Header))`. If the JWS Protected Header is not present (which can only happen when using the JWS JSON Serialization and no "protected" member is present), let this value be the empty string.
5. Compute the JWS Signature in the manner defined for the particular algorithm being used over the JWS Signing Input `ASCII(BASE64URL(UTF8(JWS Protected Header)) || '.' || BASE64URL(JWS Payload))`. The "alg" (algorithm) Header Parameter MUST be present in the JWS Header, with the algorithm value accurately representing the algorithm used to construct the JWS Signature.
6. Compute the encoded signature value `BASE64URL(JWS Signature)`.
7. These three encoded values are used in both the JWS Compact Serialization and the JWS JSON Serialization representations.
8. If the JWS JSON Serialization is being used, repeat this process for each digital signature or MAC value being applied.
9. Create the desired serialized output. The JWS Compact Serialization of this result is `BASE64URL(UTF8(JWS Protected Header)) || '.' || BASE64URL(JWS Payload) || '.' || BASE64URL(JWS Signature)`. The JWS JSON Serialization is described in Section 7.2.

5.2. Message Signature or MAC Validation

When validating a JWS, the following steps MUST be taken. The order of the steps is not significant in cases where there are no dependencies between the inputs and outputs of the steps. If any of the listed steps fails, then the signature or MAC cannot be validated.

It is an application decision which signatures, MACs, or plaintext values must successfully validate for the JWS to be accepted. In some cases, all must successfully validate or the JWS will be rejected. In other cases, only a specific signature, MAC, or plaintext value needs to be successfully validated. However, in all cases, at least one signature, MAC, or plaintext value MUST successfully validate or the JWS MUST be rejected.

1. Parse the JWS representation to extract the serialized values for the components of the JWS -- when using the JWS Compact Serialization, the base64url encoded representations of the JWS Protected Header, the JWS Payload, and the JWS Signature, and when using the JWS JSON Serialization, also the unencoded JWS Unprotected Header value. When using the JWS Compact Serialization, the JWS Protected Header, the JWS Payload, and the JWS Signature are represented as base64url encoded values in that order, separated by two period ('.') characters. The JWS JSON Serialization is described in Section 7.2.
2. The encoded representation of the JWS Protected Header MUST be successfully base64url decoded following the restriction that no padding characters have been used.
3. The resulting UTF8 encoded JWS Protected Header MUST be a completely valid JSON object conforming to RFC 4627 [RFC4627].
4. If using the JWS Compact Serialization, let the JWS Header be the JWS Protected Header; otherwise, when using the JWS JSON Serialization, let the JWS Header be the union of the members of the corresponding JWS Protected Header and JWS Unprotected Header, all of which must be completely valid JSON objects.
5. The resulting JWS Header MUST NOT contain duplicate Header Parameter names. When using the JWS JSON Serialization, this restriction includes that the same Header Parameter name also MUST NOT occur in distinct JSON Text Object values that together comprise the JWS Header.
6. The resulting JWS Header MUST be validated to only include parameters and values whose syntax and semantics are both understood and supported or that are specified as being ignored when not understood.
7. The encoded representation of the JWS Payload MUST be successfully base64url decoded following the restriction that no padding characters have been used.
8. The encoded representation of the JWS Signature MUST be successfully base64url decoded following the restriction that no padding characters have been used.
9. The JWS Signature MUST be successfully validated against the JWS Signing Input ASCII(BASE64URL(UTF8(JWS Protected Header)) || '.' || BASE64URL(JWS Payload)) in the manner defined for the algorithm being used, which MUST be accurately represented by the value of the "alg" (algorithm) Header Parameter, which MUST

- be present.
10. If the JWS JSON Serialization is being used, repeat this process for each digital signature or MAC value contained in the representation.

5.3. String Comparison Rules

Processing a JWS inevitably requires comparing known strings to values in JSON objects. For example, in checking what the algorithm is, the Unicode string encoding "alg" will be checked against the member names in the JWS Header to see if there is a matching Header Parameter name. A similar process occurs when determining if the value of the "alg" Header Parameter represents a supported algorithm.

Comparisons between JSON strings and other Unicode strings MUST be performed as specified below:

1. Remove any JSON escaping from the input JSON string and convert the string into a sequence of Unicode code points.
2. Likewise, convert the string to be compared against into a sequence of Unicode code points.
3. Unicode Normalization [USA15] MUST NOT be applied at any point to either the JSON string or to the string it is to be compared against.
4. Comparisons between the two strings MUST be performed as a Unicode code point to code point equality comparison. (Note that values that originally used different Unicode encodings (UTF-8, UTF-16, etc.) may result in the same code point values.)

Also, see the JSON security considerations in Section 9.2 and the Unicode security considerations in Section 9.3.

6. Key Identification

It is necessary for the recipient of a JWS to be able to determine the key that was employed for the digital signature or MAC operation. The key employed can be identified using the Header Parameter methods described in Section 4.1 or can be identified using methods that are outside the scope of this specification. Specifically, the Header Parameters "jku", "jwk", "x5u", "x5t", "x5c", and "kid" can be used to identify the key used. These Header Parameters MUST be integrity protected if the information that they convey is to be utilized in a trust decision.

The sender SHOULD include sufficient information in the Header Parameters to identify the key used, unless the application uses another means or convention to determine the key used. Validation of the signature or MAC fails when the algorithm used requires a key

(which is true of all algorithms except for "none") and the key used cannot be determined.

The means of exchanging any shared symmetric keys used is outside the scope of this specification.

7. Serializations

JWS objects use one of two serializations, the JWS Compact Serialization or the JWS JSON Serialization. For general-purpose implementations, both the JWS Compact Serialization and JWS JSON Serialization support for the single signature or MAC value case are mandatory to implement; support for multiple signatures and/or MAC values is OPTIONAL. Special-purpose implementations are permitted to implement only a single serialization, if that meets the needs of the targeted use cases.

7.1. JWS Compact Serialization

The JWS Compact Serialization represents digitally signed or MACed content as a compact URL-safe string. This string is `BASE64URL(UTF8(JWS Protected Header)) || '.' || BASE64URL(JWS Payload) || '.' || BASE64URL(JWS Signature)`. Only one signature/MAC is supported by the JWS Compact Serialization.

7.2. JWS JSON Serialization

The JWS JSON Serialization represents digitally signed or MACed content as a JSON object. Unlike the JWS Compact Serialization, content using the JWS JSON Serialization can be secured with more than one digital signature and/or MAC value.

The representation is closely related to that used in the JWS Compact Serialization, with the following differences for the JWS JSON Serialization:

- o Values in the JWS JSON Serialization are represented as members of a JSON object, rather than as base64url encoded strings separated by period ('.') characters. (However binary values and values that are integrity protected are still base64url encoded.)
- o The value `BASE64URL(JWS Payload)` is stored in the "payload" member.
- o There can be multiple signature and/or MAC values, rather than just one. A JSON array in the "signatures" member is used to hold values that are specific to a particular signature or MAC computation, with one array element per signature/MAC represented. These array elements are JSON objects.

- o Each value `BASE64URL(JWS Signature)`, if non-empty, is stored in the "signature" member of a JSON object that is an element of the "signatures" array.
- o Each value `BASE64URL(UTF8(JWS Protected Header))`, if non-empty, is stored in the "protected" member of the corresponding element of the "signatures" array.
- o Each JWS Unprotected Header value, if non-empty, is stored in the "header" member of the corresponding element of the "signatures" array. If present, a JWS Unprotected Header value is represented as an unencoded JSON Text Object, rather than as a string.
- o The Header Parameter values used when creating or validating individual signature or MAC values are the union of the two sets of Header Parameter values that may be present: (1) the JWS Protected Header values represented in the "protected" member of the signature/MAC's array element, and (2) the JWS Unprotected Header values in the "header" member of the signature/MAC's array element. The union of these sets of Header Parameters comprises the JWS Header. The Header Parameter names in the two locations MUST be disjoint.

The syntax of a JWS using the JWS JSON Serialization is as follows:

```
{
  "payload": "<payload contents>"
  "signatures": [
    {
      "protected": "<integrity-protected header 1 contents>",
      "header": "<non-integrity-protected header 1 contents>",
      "signature": "<signature 1 contents>"
    },
    ...
    {
      "protected": "<integrity-protected header N contents>",
      "header": "<non-integrity-protected header N contents>",
      "signature": "<signature N contents>"
    }
  ]
}
```

Of these members, only the "payload", "signatures", and "signature" members MUST be present. At least one of the "protected" and "header" members MUST be present for each signature/MAC computation so that an "alg" Header Parameter value is conveyed.

The contents of the JWS Payload and JWS Signature values are exactly as defined in the rest of this specification. They are interpreted and validated in the same manner, with each corresponding JWS Signature and set of Header Parameter values being created and validated together. The JWS Header values used are the union of the Header Parameters in the corresponding JWS Protected Header and JWS Unprotected Header values, as described earlier.

Each JWS Signature value is computed on the JWS Signing Input using

the parameters of the corresponding JWS Header value in the same manner as for the JWS Compact Serialization. This has the desirable property that each JWS Signature value represented in the "signatures" array is identical to the value that would have been computed for the same parameter in the JWS Compact Serialization, provided that the JWS Protected Header value for that signature/MAC computation (which represents the integrity-protected Header Parameter values) matches that used in the JWS Compact Serialization.

See Appendix A.6 for an example of computing a JWS using the JWS JSON Serialization.

8. 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).

8.1. JSON Web Signature and Encryption Header Parameters Registry

This specification establishes the IANA JSON Web Signature and Encryption Header Parameters registry for JWS and JWE Header Parameter names. The registry records the Header Parameter name and a reference to the specification that defines it. The same Header Parameter name MAY be registered multiple times, provided that the parameter usage is compatible between the specifications. Different registrations of the same Header Parameter name will typically use different Header Parameter Usage Location(s) values.

8.1.1. Registration Template

Header Parameter 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.

Header Parameter Usage Location(s):

The Header Parameter usage locations, which should be one or more of the values "JWS" or "JWE".

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.

8.1.2. Initial Registry Contents

This specification registers the Header Parameter names defined in Section 4.1 in this registry.

- o Header Parameter Name: "alg"
- o Header Parameter Usage Location(s): JWS
- o Change Controller: IESG
- o Specification Document(s): Section 4.1.1 of [[this document]]

- o Header Parameter Name: "jku"
- o Header Parameter Usage Location(s): JWS
- o Change Controller: IESG
- o Specification Document(s): Section 4.1.2 of [[this document]]

- o Header Parameter Name: "jwk"
- o Header Parameter Usage Location(s): JWS
- o Change Controller: IESG
- o Specification document(s): Section 4.1.3 of [[this document]]

- o Header Parameter Name: "x5u"
- o Header Parameter Usage Location(s): JWS
- o Change Controller: IESG
- o Specification Document(s): Section 4.1.4 of [[this document]]

- o Header Parameter Name: "x5t"
- o Header Parameter Usage Location(s): JWS
- o Change Controller: IESG
- o Specification Document(s): Section 4.1.5 of [[this document]]

- o Header Parameter Name: "x5c"
- o Header Parameter Usage Location(s): JWS
- o Change Controller: IESG
- o Specification Document(s): Section 4.1.6 of [[this document]]

- o Header Parameter Name: "kid"
- o Header Parameter Usage Location(s): JWS
- o Change Controller: IESG
- o Specification Document(s): Section 4.1.7 of [[this document]]

- o Header Parameter Name: "typ"
- o Header Parameter Usage Location(s): JWS
- o Change Controller: IESG
- o Specification Document(s): Section 4.1.8 of [[this document]]

- o Header Parameter Name: "cty"
- o Header Parameter Usage Location(s): JWS
- o Change Controller: IESG
- o Specification Document(s): Section 4.1.9 of [[this document]]

- o Header Parameter Name: "crit"

- o Header Parameter Usage Location(s): JWS
- o Change Controller: IESG
- o Specification Document(s): Section 4.1.10 of [[this document]]

8.2. Media Type Registration

8.2.1. Registry Contents

This specification registers the "application/jose" Media Type [RFC2046] in the MIME Media Types registry [IANA.MediaType], which can be used to indicate that the content is a JWS or JWE object using the JWS Compact Serialization or the JWE Compact Serialization and the "application/jose+json" Media Type in the MIME Media Types registry, which can be used to indicate that the content is a JWS or JWE object using the JWS JSON Serialization or the JWE JSON Serialization.

- o Type name: application
 - o Subtype name: jose
 - o Required parameters: n/a
 - o Optional parameters: n/a
 - o Encoding considerations: 8bit; application/jose values are encoded as a series of base64url encoded values (some of which may be the empty string) separated by period ('.') characters.
 - o Security considerations: See the Security Considerations section of [[this document]]
 - o Interoperability considerations: n/a
 - o Published specification: [[this document]]
 - o Applications that use this media type: OpenID Connect, Mozilla Persona, Salesforce, Google, numerous others that use signed JWTs
 - o Additional information: Magic number(s): n/a, File extension(s): n/a, Macintosh file type code(s): n/a
 - o Person & email address to contact for further information: Michael B. Jones, mbj@microsoft.com
 - o Intended usage: COMMON
 - o Restrictions on usage: none
 - o Author: Michael B. Jones, mbj@microsoft.com
 - o Change Controller: IESG
-
- o Type name: application
 - o Subtype name: jose+json
 - o Required parameters: n/a
 - o Optional parameters: n/a
 - o Encoding considerations: 8bit; application/jose+json values are represented as a JSON Object; UTF-8 encoding SHOULD be employed for the JSON object.

- o Security considerations: See the Security Considerations section of [[this document]]
- o Interoperability considerations: n/a
- o Published specification: [[this document]]
- o Applications that use this media type: TBD
- o Additional information: Magic number(s): n/a, File extension(s): n/a, Macintosh file type code(s): n/a
- o Person & email address to contact for further information: Michael B. Jones, mbj@microsoft.com
- o Intended usage: COMMON
- o Restrictions on usage: none
- o Author: Michael B. Jones, mbj@microsoft.com
- o Change Controller: IESG

9. Security Considerations

9.1. Cryptographic 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 concerns are listed here.

All the security considerations in XML DSIG 2.0 [W3C.CR-xmlsig-core2-20120124], also apply to this specification, other than those that are XML specific. Likewise, many of the best practices documented in XML Signature Best Practices [W3C.WD-xmlsig-bestpractices-20110809] also apply to this specification, other than those that are XML specific.

Keys are only as strong as the amount of entropy used to generate them. A minimum of 128 bits of entropy should be used for all keys, and depending upon the application context, more may be required. In particular, it may be difficult to generate sufficiently random values in some browsers and application environments.

Creators of JWSs should not allow third parties to insert arbitrary content into the message without adding entropy not controlled by the third party.

When utilizing TLS to retrieve information, the authority providing the resource MUST be authenticated and the information retrieved MUST be free from modification.

When cryptographic algorithms are implemented in such a way that successful operations take a different amount of time than unsuccessful operations, attackers may be able to use the time difference to obtain information about the keys employed. Therefore, such timing differences must be avoided.

A SHA-1 hash is used when computing "x5t" (x.509 certificate thumbprint) values, for compatibility reasons. Should an effective means of producing SHA-1 hash collisions be developed, and should an attacker wish to interfere with the use of a known certificate on a given system, this could be accomplished by creating another certificate whose SHA-1 hash value is the same and adding it to the certificate store used by the intended victim. A prerequisite to this attack succeeding is the attacker having write access to the intended victim's certificate store.

If, in the future, certificate thumbprints need to be computed using hash functions other than SHA-1, it is suggested that additional related Header Parameters be defined for that purpose. For example, it is suggested that a new "x5t#S256" (X.509 Certificate Thumbprint using SHA-256) Header Parameter could be defined and used.

9.2. JSON Security Considerations

Strict JSON validation is a security requirement. If malformed JSON is received, then the intent of the sender is impossible to reliably discern. Ambiguous and potentially exploitable situations could arise if the JSON parser used does not reject malformed JSON syntax.

Section 2.2 of the JavaScript Object Notation (JSON) specification [RFC4627] states "The names within an object SHOULD be unique", whereas this specification states that "Header Parameter names within this object MUST be unique; recipients MUST either reject JWSs with duplicate Header Parameter names or use a JSON parser that returns only the lexically last duplicate member name, as specified in Section 15.12 (The JSON Object) of ECMA Script 5.1 [ECMAScript]". Thus, this specification requires that the Section 2.2 "SHOULD" be treated as a "MUST" by senders and that it be either treated as a "MUST" or in the manner specified in ECMA Script 5.1 by receivers. Ambiguous and potentially exploitable situations could arise if the JSON parser used does not enforce the uniqueness of member names or returns an unpredictable value for duplicate member names.

Some JSON parsers might not reject input that contains extra significant characters after a valid input. For instance, the input `{"tag":"value"}ABCD` contains a valid JSON object followed by the extra characters "ABCD". Such input MUST be rejected in its entirety.

9.3. Unicode Comparison Security Considerations

Header Parameter names and algorithm names are Unicode strings. For security reasons, the representations of these names must be compared verbatim after performing any escape processing (as per RFC 4627 [RFC4627], Section 2.5). This means, for instance, that these JSON strings must compare as being equal ("sig", "\u0073ig"), whereas these must all compare as being not equal to the first set or to each other ("SIG", "Sig", "si\u0047").

JSON strings can contain characters outside the Unicode Basic Multilingual Plane. For instance, the G clef character (U+1D11E) may be represented in a JSON string as "\uD834\uDD1E". Ideally, JWS implementations SHOULD ensure that characters outside the Basic Multilingual Plane are preserved and compared correctly; alternatively, if this is not possible due to these characters exercising limitations present in the underlying JSON implementation, then input containing them MUST be rejected.

9.4. TLS Requirements

Implementations MUST support TLS. Which version(s) ought to be implemented will vary over time, and depend on the widespread deployment and known security vulnerabilities at the time of implementation. At the time of this writing, TLS version 1.2 [RFC5246] is the most recent version, but has very limited actual deployment, and might not be readily available in implementation toolkits. TLS version 1.0 [RFC2246] is the most widely deployed version, and will give the broadest interoperability.

To protect against information disclosure and tampering, confidentiality protection MUST be applied using TLS with a ciphersuite that provides confidentiality and integrity protection.

Whenever TLS is used, a TLS server certificate check MUST be performed, per RFC 6125 [RFC6125].

10. References

10.1. Normative References

[ECMAScript]

Ecma International, "ECMAScript Language Specification, 5.1 Edition", ECMA 262, June 2011.

[IANA.MediaType]

Internet Assigned Numbers Authority (IANA), "MIME Media

Types", 2005.

[ITU.X690.1994]

International Telecommunications Union, "Information Technology - ASN.1 encoding rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER)", ITU-T Recommendation X.690, 1994.

[JWA]

Jones, M., "JSON Web Algorithms (JWA)", draft-ietf-jose-json-web-algorithms (work in progress), October 2013.

[JWK]

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

[RFC1421]

Linn, J., "Privacy Enhancement for Internet Electronic Mail: Part I: Message Encryption and Authentication Procedures", RFC 1421, February 1993.

[RFC2045]

Freed, N. and N. Borenstein, "Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies", RFC 2045, November 1996.

[RFC2046]

Freed, N. and N. Borenstein, "Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types", RFC 2046, November 1996.

[RFC2119]

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

[RFC2246]

Dierks, T. and C. Allen, "The TLS Protocol Version 1.0", RFC 2246, January 1999.

[RFC2818]

Rescorla, E., "HTTP Over TLS", RFC 2818, May 2000.

[RFC3629]

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

[RFC3986]

Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax", STD 66, RFC 3986, January 2005.

[RFC4627]

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

[RFC4648]

Josefsson, S., "The Base16, Base32, and Base64 Data

Encodings", RFC 4648, October 2006.

- [RFC5226] Narten, T. and H. Alvestrand, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 5226, May 2008.
- [RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, August 2008.
- [RFC5280] Cooper, D., Santesson, S., Farrell, S., Boeyen, S., Housley, R., and W. Polk, "Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile", RFC 5280, May 2008.
- [RFC6125] Saint-Andre, P. and J. Hodges, "Representation and Verification of Domain-Based Application Service Identity within Internet Public Key Infrastructure Using X.509 (PKIX) Certificates in the Context of Transport Layer Security (TLS)", RFC 6125, March 2011.
- [USA15] Davis, M., Whistler, K., and M. Deurst, "Unicode Normalization Forms", Unicode Standard Annex 15, 09 2009.
- [USASCII] American National Standards Institute, "Coded Character Set -- 7-bit American Standard Code for Information Interchange", ANSI X3.4, 1986.
- [W3C.WD-xmlsig-bestpractices-20110809] Datta, P. and F. Hirsch, "XML Signature Best Practices", World Wide Web Consortium WD WD-xmlsig-bestpractices-20110809, August 2011, <<http://www.w3.org/TR/2011/WD-xmlsig-bestpractices-20110809>>.

10.2. Informative References

- [CanvasApp] Facebook, "Canvas Applications", 2010.
- [JSS] Bradley, J. and N. Sakimura (editor), "JSON Simple Sign", September 2010.
- [JWE] Jones, M., Rescorla, E., and J. Hildebrand, "JSON Web Encryption (JWE)", draft-ietf-jose-json-web-encryption (work in progress), October 2013.
- [JWT] Jones, M., Bradley, J., and N. Sakimura, "JSON Web Token (JWT)", draft-ietf-oauth-json-web-token (work in progress), October 2013.

[MagicSignatures]

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

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

[W3C.CR-xmlsig-core2-20120124]

Eastlake, D., Reagle, J., Yiu, K., Solo, D., Datta, P., Hirsch, F., Cantor, S., and T. Roessler, "XML Signature Syntax and Processing Version 2.0", World Wide Web Consortium CR CR-xmlsig-core2-20120124, January 2012, <<http://www.w3.org/TR/2012/CR-xmlsig-core2-20120124>>.

Appendix A. JWS Examples

This section provides several examples of JWSs. While the first three examples all represent JSON Web Tokens (JWTs) [JWT], the payload can be any octet sequence, as shown in Appendix A.4.

A.1. Example JWS using HMAC SHA-256

A.1.1. Encoding

The following example JWS Protected Header declares that the data structure is a JSON Web Token (JWT) [JWT] and the JWS Signing Input is secured using the HMAC SHA-256 algorithm.

```
{ "typ": "JWT",  
  "alg": "HS256" }
```

The octets representing UTF8(JWS Protected Header) in this case are:

```
[123, 34, 116, 121, 112, 34, 58, 34, 74, 87, 84, 34, 44, 13, 10, 32,  
34, 97, 108, 103, 34, 58, 34, 72, 83, 50, 53, 54, 34, 125]
```

Encoding this JWS Protected Header as BASE64URL(UTF8(JWS Protected Header)) gives this value:

```
eyJ0eXAiOiJKV1QiLA0KICJhbGciOiJIUzI1NiJ9
```

The JWS Payload used in this example is the octets of the UTF-8 representation of the JSON object below. (Note that the payload can be any base64url encoded octet sequence, and need not be a base64url encoded JSON object.)

```
{ "iss": "joe",  
  "exp": 1300819380,  
  "http://example.com/is_root": true }
```

The following octet sequence, which is the UTF-8 representation of the JSON object above, is the JWS Payload:

```
[123, 34, 105, 115, 115, 34, 58, 34, 106, 111, 101, 34, 44, 13, 10,  
32, 34, 101, 120, 112, 34, 58, 49, 51, 48, 48, 56, 49, 57, 51, 56,  
48, 44, 13, 10, 32, 34, 104, 116, 116, 112, 58, 47, 47, 101, 120, 97,  
109, 112, 108, 101, 46, 99, 111, 109, 47, 105, 115, 95, 114, 111,  
111, 116, 34, 58, 116, 114, 117, 101, 125]
```

Encoding this JWS Protected Header as `BASE64URL(UTF8(JWS Protected Header))` gives this value (with line breaks for display purposes only):

```
eyJpc3MiOiJqb2UiLA0KICJleHAiOjEzMDA4MTkzODAsDQogImh0dHA6Ly9leGFT  
cGxlLnNvbS9pc19yb290Ijp0cnVlfQ
```

Combining these as `BASE64URL(UTF8(JWS Protected Header)) || '.' || BASE64URL(JWS Payload)` gives this string (with line breaks for display purposes only):

```
eyJ0eXAiOiJKV1QiLA0KICJhbGciOiJIUzI1NiJ9  
.  
eyJpc3MiOiJqb2UiLA0KICJleHAiOjEzMDA4MTkzODAsDQogImh0dHA6Ly9leGFT  
cGxlLnNvbS9pc19yb290Ijp0cnVlfQ
```

The resulting JWS Signing Input value, which is the ASCII representation of above string, is the following octet sequence:

```
[101, 121, 74, 48, 101, 88, 65, 105, 79, 105, 74, 75, 86, 49, 81,  
105, 76, 65, 48, 75, 73, 67, 74, 104, 98, 71, 99, 105, 79, 105, 74,  
73, 85, 122, 73, 49, 78, 105, 74, 57, 46, 101, 121, 74, 112, 99, 51,  
77, 105, 79, 105, 74, 113, 98, 50, 85, 105, 76, 65, 48, 75, 73, 67,  
74, 108, 101, 72, 65, 105, 79, 106, 69, 122, 77, 68, 65, 52, 77, 84,  
107, 122, 79, 68, 65, 115, 68, 81, 111, 103, 73, 109, 104, 48, 100,  
72, 65, 54, 76, 121, 57, 108, 101, 71, 70, 116, 99, 71, 120, 108, 76,  
109, 78, 118, 98, 83, 57, 112, 99, 49, 57, 121, 98, 50, 57, 48, 73,  
106, 112, 48, 99, 110, 86, 108, 102, 81]
```

HMACs are generated using keys. This example uses the symmetric key represented in JSON Web Key [JWK] format below (with line breaks for display purposes only):

```
{ "kty": "oct",
  "k": "AyMlSysPpbyDfgZld3umjlqzKObwVMkoqQ-EstJQLr_T-1qS0gZH75
      aKtMN3Yj0iPS4hcgUuTwjAzZr1Z9CAow"
}
```

Running the HMAC SHA-256 algorithm on the JWS Signing Input with this key yields this JWS Signature octet sequence:

```
[116, 24, 223, 180, 151, 153, 224, 37, 79, 250, 96, 125, 216, 173,
187, 186, 22, 212, 37, 77, 105, 214, 191, 240, 91, 88, 5, 88, 83,
132, 141, 121]
```

Encoding this JWS Signature as BASE64URL(JWS Signature) gives this value:

```
dBjftJeZ4CVP-mB92K27uhbUJU1plr_wWlgFWFOEjXk
```

Concatenating these values in the order Header.Payload.Signature with period ('.') characters between the parts yields this complete JWS representation using the JWS Compact Serialization (with line breaks for display purposes only):

```
eyJ0eXAiOiJKV1QiLA0KICJhbGciOiJIUzI1NiJ9
.
eyJpc3MiOiJqb2UiLA0KICJleHAiOiJlZMDA4MTkzODAsDQogImh0dHA6Ly9leGFT
cGx1LnVkbS9pc19yb290Ijpb0cnVlfQ
.
dBjftJeZ4CVP-mB92K27uhbUJU1plr_wWlgFWFOEjXk
```

A.1.2. Validating

Since the "alg" Header Parameter is "HS256", we validate the HMAC SHA-256 value contained in the JWS Signature.

To validate the HMAC value, we repeat the previous process of using the correct key and the JWS Signing Input as input to the HMAC SHA-256 function and then taking the output and determining if it matches the JWS Signature. If it matches exactly, the HMAC has been validated.

A.2. Example JWS using RSASSA-PKCS-v1_5 SHA-256

A.2.1. Encoding

The JWS Protected Header in this example is different from the previous example in two ways: First, because a different algorithm is being used, the "alg" value is different. Second, for illustration purposes only, the optional "typ" parameter is not used. (This

difference is not related to the algorithm employed.) The JWS Protected Header used is:

```
{"alg": "RS256"}
```

The octets representing UTF8(JWS Protected Header) in this case are:

```
[123, 34, 97, 108, 103, 34, 58, 34, 82, 83, 50, 53, 54, 34, 125]
```

Encoding this JWS Protected Header as BASE64URL(UTF8(JWS Protected Header)) gives this value:

```
eyJhbGciOiJSUzI1NiJ9
```

The JWS Payload used in this example, which follows, is the same as in the previous example. Since the BASE64URL(JWS Payload) value will therefore be the same, its computation is not repeated here.

```
{"iss": "joe",  
 "exp": 1300819380,  
 "http://example.com/is_root": true}
```

Combining these as BASE64URL(UTF8(JWS Protected Header)) || '.' || BASE64URL(JWS Payload) gives this string (with line breaks for display purposes only):

```
eyJhbGciOiJSUzI1NiJ9  
.  
eyJpc3MiOiJqb2UiLA0KICJleHAiOjEzMDA4MTkzODAsDQogImh0dHA6Ly9leGft  
cGxlImNvbS9pc19yb290Ijp0cnVlfQ
```

The resulting JWS Signing Input value, which is the ASCII representation of above string, is the following octet sequence:

```
[101, 121, 74, 104, 98, 71, 99, 105, 79, 105, 74, 83, 85, 122, 73,  
49, 78, 105, 74, 57, 46, 101, 121, 74, 112, 99, 51, 77, 105, 79, 105,  
74, 113, 98, 50, 85, 105, 76, 65, 48, 75, 73, 67, 74, 108, 101, 72,  
65, 105, 79, 106, 69, 122, 77, 68, 65, 52, 77, 84, 107, 122, 79, 68,  
65, 115, 68, 81, 111, 103, 73, 109, 104, 48, 100, 72, 65, 54, 76,  
121, 57, 108, 101, 71, 70, 116, 99, 71, 120, 108, 76, 109, 78, 118,  
98, 83, 57, 112, 99, 49, 57, 121, 98, 50, 57, 48, 73, 106, 112, 48,  
99, 110, 86, 108, 102, 81]
```

This example uses the RSA key represented in JSON Web Key [JWK] format below (with line breaks for display purposes only):

```
{ "kty": "RSA",
  "n": "ofgWCuLjybRlzo0tZWJjNiuSfb4p4fAkd_wWJcyQoTbji9k0l8W26mPddx
HmfHQp-Vaw-4qPCJrcS2mJPMEzPlPt0Bm4d4QlL-yRT-SFd2lZS-pCgNMs
DlW_YpRPEwOWvG6b32690r2jZ47soMZo9wGzjb_7OMg0LOL-bSf63kpaSH
SXndS5z5rexMdbYUsLA9e-KXBdQOS-UTo7WTBEMa2R2CapHg665xsmtDv
MTBQY4uDZlxb3qCo5ZwKh9kG4LT6_I5IhlJH7aGhyxXFvUK-DWNmoudF8
NAco9_h9iaGNj8q2ethFkMLs9l1kzk2PacDTW9gb54h4FRWyuXpoQ",
  "e": "AQAB",
  "d": "Eq5xpGnNCivDflJsRQBxHx1hDr1k6UlwE2JZD50LpXyWPEAep88vLNO97I
j1A7_GQ5sLKMgvfTeXZx9SE-7YwVol2NXOoAJe46sui395IW_GO-pWJ100
BktGoVen2bKVRUCgu-GjBVaYLU6f3l9kJfFNS3E0QbVdxzubSu3Mkqzjkn
439X0M_V51gfpRLI9JYanrC4D4qAdGcopV_0ZHHZQlBjudU2QvXt4ehNYT
CBR6XCLQUShbljuUO1ZdiYoFaFQT5Tw8bGUL_x_jTj3ccPDVZFD9pIuhLh
BOneufuBiB4cS98l2SR_RQyGWSeWjnczT0QU91plDhOVRuOopznQ"
}
```

The RSA private key is then passed to the RSA signing function, which also takes the hash type, SHA-256, and the JWS Signing Input as inputs. The result of the digital signature is an octet sequence, which represents a big endian integer. In this example, it is:

```
[112, 46, 33, 137, 67, 232, 143, 209, 30, 181, 216, 45, 191, 120, 69,
243, 65, 6, 174, 27, 129, 255, 247, 115, 17, 22, 173, 209, 113, 125,
131, 101, 109, 66, 10, 253, 60, 150, 238, 221, 115, 162, 102, 62, 81,
102, 104, 123, 0, 11, 135, 34, 110, 1, 135, 237, 16, 115, 249, 69,
229, 130, 173, 252, 239, 22, 216, 90, 121, 142, 232, 198, 109, 219,
61, 184, 151, 91, 23, 208, 148, 2, 190, 237, 213, 217, 217, 112, 7,
16, 141, 178, 129, 96, 213, 248, 4, 12, 167, 68, 87, 98, 184, 31,
190, 127, 249, 217, 46, 10, 231, 111, 36, 242, 91, 51, 187, 230, 244,
74, 230, 30, 177, 4, 10, 203, 32, 4, 77, 62, 249, 18, 142, 212, 1,
48, 121, 91, 212, 189, 59, 65, 238, 202, 208, 102, 171, 101, 25, 129,
253, 228, 141, 247, 127, 55, 45, 195, 139, 159, 175, 221, 59, 239,
177, 139, 93, 163, 204, 60, 46, 176, 47, 158, 58, 65, 214, 18, 202,
173, 21, 145, 18, 115, 160, 95, 35, 185, 232, 56, 250, 175, 132, 157,
105, 132, 41, 239, 90, 30, 136, 121, 130, 54, 195, 212, 14, 96, 69,
34, 165, 68, 200, 242, 122, 122, 45, 184, 6, 99, 209, 108, 247, 202,
234, 86, 222, 64, 92, 178, 33, 90, 69, 178, 194, 85, 102, 181, 90,
193, 167, 72, 160, 112, 223, 200, 163, 42, 70, 149, 67, 208, 25, 238,
251, 71]
```

Encoding the signature as BASE64URL(JWS Signature) produces this value (with line breaks for display purposes only):

```
cC4hiUPoj9Eetdgtv3hF80EGrhuB__dzERat0XF9g2VtQgr9PJbu3XOizj5RZmh7
AAuHIm4Bh-0Qc_lF5YKt_08W2Fp5jujGbdS9uJdbF9CUAr7tldnZcAcQjbKBYNX4
BAynRFdiuB--f_nZLgrnbyTyWzO75vRK5h6xBarLIARNPvkSjtQBMHlblL07Qe7K
0GarZRmB_eSN9383LcOLn6_dO--xil2jzDwusC-eOkHWesqtFZESc6BfI7noOPqv
hJlphCnvWh6IeYI2w9QOYEUIpUTI8np6LbgGY9Fs98rqVt5AXLIhWkWywLVmtVrB
```

```
p0igcN_IoypGlUPQGe77Rw
```

Concatenating these values in the order Header.Payload.Signature with period ('.') characters between the parts yields this complete JWS representation using the JWS Compact Serialization (with line breaks for display purposes only):

```
eyJhbGciOiJSUzI1NiJ9
.
eyJpc3MiOiJqb2UiLA0KICJleHAiOjEzMDA4MTkzODAsDQogImh0dHA6Ly9leGft
cGx1LnNvbm9pc19yb290Ijpb0cnVlfQ
.
cC4hiUPoj9Eetdgtv3hF80EGrhuB__dzERat0XF9g2VtQgr9PJbu3X0iZj5RZmh7
AAuHIm4Bh-0Qc_lF5YKt_08W2Fp5jujGbds9uJdbF9CUAr7tldnZcAcQjbKBYNX4
BAynRFdiuB--f_nZLgrnbyTyWz075vRK5h6xBArLIARNPvkSjtQBMHlb1L07Qe7K
0GarZRmB_eSN9383LcOLn6_dO--xi12jzDwusC-eOkHWesqtFZEsc6BfI7noOPqv
hJlphCnvWh6IeYI2w9QOYEUIpUTI8np6LbgGY9Fs98rqVt5AXLIhWkWyw1VmtVrB
p0igcN_IoypGlUPQGe77Rw
```

A.2.2. Validating

Since the "alg" Header Parameter is "RS256", we validate the RSASSA-PKCS-v1_5 SHA-256 digital signature contained in the JWS Signature.

Validating the JWS Signature is a little different from the previous example. We pass (n, e), JWS Signature, and the JWS Signing Input to an RSASSA-PKCS-v1_5 signature verifier that has been configured to use the SHA-256 hash function.

A.3. Example JWS using ECDSA P-256 SHA-256

A.3.1. Encoding

The JWS Protected Header for this example differs from the previous example because a different algorithm is being used. The JWS Protected Header used is:

```
{"alg":"ES256"}
```

The octets representing UTF8(JWS Protected Header) in this case are:

```
[123, 34, 97, 108, 103, 34, 58, 34, 69, 83, 50, 53, 54, 34, 125]
```

Encoding this JWS Protected Header as BASE64URL(UTF8(JWS Protected Header)) gives this value:

```
eyJhbGciOiJFUzI1NiJ9
```

The JWS Payload used in this example, which follows, is the same as in the previous examples. Since the `BASE64URL(JWS Payload)` value will therefore be the same, its computation is not repeated here.

```
{ "iss": "joe",
  "exp": 1300819380,
  "http://example.com/is_root": true }
```

Combining these as `BASE64URL(UTF8(JWS Protected Header)) || '.' || BASE64URL(JWS Payload)` gives this string (with line breaks for display purposes only):

```
eyJhbGciOiJIJFZlI1NiJ9
.
eyJpc3MiOiJqb2UiLA0KICJleHAiOjEzMDA4MTkzODAsDQogImh0dHA6Ly9leGFt
cGxlLnNvbS9pc19yb290Ijpb0cnVlfiQ
```

The resulting JWS Signing Input value, which is the ASCII representation of above string, is the following octet sequence:

```
[101, 121, 74, 104, 98, 71, 99, 105, 79, 105, 74, 70, 85, 122, 73,
49, 78, 105, 74, 57, 46, 101, 121, 74, 112, 99, 51, 77, 105, 79, 105,
74, 113, 98, 50, 85, 105, 76, 65, 48, 75, 73, 67, 74, 108, 101, 72,
65, 105, 79, 106, 69, 122, 77, 68, 65, 52, 77, 84, 107, 122, 79, 68,
65, 115, 68, 81, 111, 103, 73, 109, 104, 48, 100, 72, 65, 54, 76,
121, 57, 108, 101, 71, 70, 116, 99, 71, 120, 108, 76, 109, 78, 118,
98, 83, 57, 112, 99, 49, 57, 121, 98, 50, 57, 48, 73, 106, 112, 48,
99, 110, 86, 108, 102, 81]
```

This example uses the elliptic curve key represented in JSON Web Key [JWK] format below:

```
{ "kty": "EC",
  "crv": "P-256",
  "x": "f83OJ3D2xF1Bg8vub9tLe1gHMzV76e8Tus9uPHvRVEU",
  "y": "x_FeZRu9m36HLN_tue659LNpXW6pCyStikYjKIWI5a0",
  "d": "jpsQnnGQmL-YBIffH1136cspYG6-0iY7X1fCE9-E9LI"
}
```

The ECDSA private part `d` is then passed to an ECDSA signing function, which also takes the curve type, P-256, the hash type, SHA-256, and the JWS Signing Input as inputs. The result of the digital signature is the EC point (R, S), where R and S are unsigned integers. In this example, the R and S values, given as octet sequences representing big endian integers are:

Result Name	Value
R	[14, 209, 33, 83, 121, 99, 108, 72, 60, 47, 127, 21, 88, 7, 212, 2, 163, 178, 40, 3, 58, 249, 124, 126, 23, 129, 154, 195, 22, 158, 166, 101]
S	[197, 10, 7, 211, 140, 60, 112, 229, 216, 241, 45, 175, 8, 74, 84, 128, 166, 101, 144, 197, 242, 147, 80, 154, 143, 63, 127, 138, 131, 163, 84, 213]

The JWS Signature is the value R || S. Encoding the signature as BASE64URL(JWS Signature) produces this value (with line breaks for display purposes only):

```
DtEhU3ljbEg8L38VWafUAqOyKAM6-Xx-F4GawxaepmXFCgfTjDxw5djxLa8ISlSA
pmWQxfKTUJqPP3-Kg6NUlQ
```

Concatenating these values in the order Header.Payload.Signature with period ('.') characters between the parts yields this complete JWS representation using the JWS Compact Serialization (with line breaks for display purposes only):

```
eyJhbGciOiJFUzI1NiJ9
.
eyJpc3MiOiJqb2UiLA0KICJleHAiOjEzMDA4MTkzODAsDQogImh0dHA6Ly9leGFt
cGx1LnNvbS9pc19yb290Ijp0cnVlfQ
.
DtEhU3ljbEg8L38VWafUAqOyKAM6-Xx-F4GawxaepmXFCgfTjDxw5djxLa8ISlSA
pmWQxfKTUJqPP3-Kg6NUlQ
```

A.3.2. Validating

Since the "alg" Header Parameter is "ES256", we validate the ECDSA P-256 SHA-256 digital signature contained in the JWS Signature.

Validating the JWS Signature is a little different from the first example. We need to split the 64 member octet sequence of the JWS Signature into two 32 octet sequences, the first R and the second S. We then pass (x, y), (R, S) and the JWS Signing Input to an ECDSA signature verifier that has been configured to use the P-256 curve with the SHA-256 hash function.

A.4. Example JWS using ECDSA P-521 SHA-512

A.4.1. Encoding

The JWS Protected Header for this example differs from the previous example because different ECDSA curves and hash functions are used. The JWS Protected Header used is:

```
{"alg":"ES512"}
```

The octets representing UTF8(JWS Protected Header) in this case are:

```
[123, 34, 97, 108, 103, 34, 58, 34, 69, 83, 53, 49, 50, 34, 125]
```

Encoding this JWS Protected Header as BASE64URL(UTF8(JWS Protected Header)) gives this value:

```
eyJhbGciOiJFUzUxMiJ9
```

The JWS Payload used in this example, is the ASCII string "Payload". The representation of this string is the octet sequence:

```
[80, 97, 121, 108, 111, 97, 100]
```

Encoding this JWS Payload as BASE64URL(JWS Payload) gives this value:

```
UGF5bG9hZA
```

Combining these as BASE64URL(UTF8(JWS Protected Header)) || '.' || BASE64URL(JWS Payload) gives this string (with line breaks for display purposes only):

```
eyJhbGciOiJFUzUxMiJ9.UGF5bG9hZA
```

The resulting JWS Signing Input value, which is the ASCII representation of above string, is the following octet sequence:

```
[101, 121, 74, 104, 98, 71, 99, 105, 79, 105, 74, 70, 85, 122, 85,  
120, 77, 105, 74, 57, 46, 85, 71, 70, 53, 98, 71, 57, 104, 90, 65]
```

This example uses the elliptic curve key represented in JSON Web Key [JWK] format below (with line breaks for display purposes only):

```
{ "kty": "EC",
  "crv": "P-521",
  "x": "AekpBQ8ST8a8VcfVOTNl353vSrDCLLJXmPk06wTjxr rjcBpXp5EOnYG_
    NjFZ6OvLfv1jSfs9tsz4qUxcWceqwQGk",
  "y": "ADSmRA43Z1DSNx_RvcLI87cdL0716jQyyBXMoxVg_l2Th-x3S1WDhjd1
    y79ajL4Kkd0AZMaZmh9ubmf63e3kyMj2",
  "d": "AY5pb7A0UfiB3RELSd64fTLOSv_jazdF7fLYyuTw8lOfRhWg6Y6rUrPA
    xerEzgdRhajnu0ferB0d53vM9mEl5j2C"
}
```

The ECDSA private part `d` is then passed to an ECDSA signing function, which also takes the curve type, P-521, the hash type, SHA-512, and the JWS Signing Input as inputs. The result of the digital signature is the EC point (R, S), where R and S are unsigned integers. In this example, the R and S values, given as octet sequences representing big endian integers are:

Result Name	Value
R	[1, 220, 12, 129, 231, 171, 194, 209, 232, 135, 233, 117, 247, 105, 122, 210, 26, 125, 192, 1, 217, 21, 82, 91, 45, 240, 255, 83, 19, 34, 239, 71, 48, 157, 147, 152, 105, 18, 53, 108, 163, 214, 68, 231, 62, 153, 150, 106, 194, 164, 246, 72, 143, 138, 24, 50, 129, 223, 133, 206, 209, 172, 63, 237, 119, 109]
S	[0, 111, 6, 105, 44, 5, 41, 208, 128, 61, 152, 40, 92, 61, 152, 4, 150, 66, 60, 69, 247, 196, 170, 81, 193, 199, 78, 59, 194, 169, 16, 124, 9, 143, 42, 142, 131, 48, 206, 238, 34, 175, 83, 203, 220, 159, 3, 107, 155, 22, 27, 73, 111, 68, 68, 21, 238, 144, 229, 232, 148, 188, 222, 59, 242, 103]

The JWS Signature is the value `R || S`. Encoding the signature as `BASE64URL(JWS Signature)` produces this value (with line breaks for display purposes only):

```
AdwMgeerwtHoh-1192160hp9wAHZFVJbLfD_UxMi70cwnZOYaRI1bKPWROc-mZZq
wqT2SI-KGDKB34XO0aw_7XdtAG8GaSwFKdCAPZgoXD2YBJZCPEX3xKpRwcd008Kp
EHwJjyqOgzDO7iKvU8vcnwNrmxYbSW9ERBXukOXolLzeO_Jn
```

Concatenating these values in the order `Header.Payload.Signature` with period ('.') characters between the parts yields this complete JWS representation using the JWS Compact Serialization (with line breaks for display purposes only):

```
eyJhbGciOiJFUzUxMiJ9
.
UGF5bG9hZA
.
AdwMgeerwtHoh-1192160hp9wAHZfVJbLfD_UxMi70cwnZOYaRI1bKPWROc-mZZq
wqT2SI-KGDKB34XO0aw_7XdtAG8GaSwFKdCAPZgoXD2YBJZCPEX3xKpRwcd008Kp
EHwJjyqOgzDO7iKvU8vcnwNrmxYbSW9ERBXukOXolLzeO_Jn
```

A.4.2. Validating

Since the "alg" Header Parameter is "ES512", we validate the ECDSA P-521 SHA-512 digital signature contained in the JWS Signature.

Validating the JWS Signature is similar to the previous example. We need to split the 132 member octet sequence of the JWS Signature into two 66 octet sequences, the first R and the second S. We then pass (x, y), (R, S) and the JWS Signing Input to an ECDSA signature verifier that has been configured to use the P-521 curve with the SHA-512 hash function.

A.5. Example Plaintext JWS

The following example JWS Protected Header declares that the encoded object is a Plaintext JWS:

```
{"alg":"none"}
```

Encoding this JWS Protected Header as `BASE64URL(UTF8(JWS Protected Header))` gives this value:

```
eyJhbGciOiJub25lIn0
```

The JWS Payload used in this example, which follows, is the same as in the previous examples. Since the `BASE64URL(JWS Payload)` value will therefore be the same, its computation is not repeated here.

```
{"iss":"joe",
 "exp":1300819380,
 "http://example.com/is_root":true}
```

The JWS Signature is the empty octet string and `BASE64URL(JWS Signature)` is the empty string.

Concatenating these parts in the order `Header.Payload.Signature` with period ('.') characters between the parts yields this complete JWS (with line breaks for display purposes only):

```
eyJhbGciOiJub251In0
.
eyJpc3MiOiJqb2UiLA0KICJleHAiOjEzMDA4MTkzODAsDQogImh0dHA6Ly9leGFT
cGx1LmNvbS9pc19yb290Ijp0cnVlfQ
.
```

A.6. Example JWS Using JWS JSON Serialization

This section contains an example using the JWS JSON Serialization. This example demonstrates the capability for conveying multiple digital signatures and/or MACs for the same payload.

The JWS Payload used in this example is the same as that used in the examples in Appendix A.2 and Appendix A.3 (with line breaks for display purposes only):

```
eyJpc3MiOiJqb2UiLA0KICJleHAiOjEzMDA4MTkzODAsDQogImh0dHA6Ly9leGFT
cGx1LmNvbS9pc19yb290Ijp0cnVlfQ
```

Two digital signatures are used in this example: the first using RSASSA-PKCS-v1_5 SHA-256 and the second using ECDSA P-256 SHA-256. For the first, the JWS Protected Header and key are the same as in Appendix A.2, resulting in the same JWS Signature value; therefore, its computation is not repeated here. For the second, the JWS Protected Header and key are the same as in Appendix A.3, resulting in the same JWS Signature value; therefore, its computation is not repeated here.

A.6.1. JWS Per-Signature Protected Headers

The JWS Protected Header value used for the first signature is:

```
{"alg":"RS256"}
```

Encoding this JWS Protected Header as `BASE64URL(UTF8(JWS Protected Header))` gives this value:

```
eyJhbGciOiJSUzI1NiJ9
```

The JWS Protected Header value used for the second signature is:

```
{"alg":"ES256"}
```

Encoding this JWS Protected Header as `BASE64URL(UTF8(JWS Protected Header))` gives this value:

```
eyJhbGciOiJFUzI1NiJ9
```

A.6.2. JWS Per-Signature Unprotected Headers

Key ID values are supplied for both keys using per-signature Header Parameters. The two values used to represent these Key IDs are:

```
{"kid": "2010-12-29"}
```

and

```
{"kid": "e9bc097a-ce51-4036-9562-d2ade882db0d"}
```

A.6.3. Complete JWS Header Values

Combining the protected and unprotected header values supplied, the JWS Header values used for the first and second signatures respectively are:

```
{"alg": "RS256",  
 "kid": "2010-12-29"}
```

and

```
{"alg": "ES256",  
 "kid": "e9bc097a-ce51-4036-9562-d2ade882db0d"}
```

A.6.4. Complete JWS JSON Serialization Representation

The complete JSON Web Signature JSON Serialization for these values is as follows (with line breaks for display purposes only):

```
{ "payload":
  "eyJpc3MiOiJqb2UiLA0KICJleHAiOjEzMDA4MTkzODAsDQogImh0dHA6Ly9leGFtcGxlLnNvbS9pc19yb290Ijpb0cnVlfQ",
  "signatures": [
    { "protected": "eyJhbGciOiJSUzI1NiJ9",
      "header": { "kid": "2010-12-29" },
      "signature": "cC4hiUPoj9Eetdgtv3hF80EGrhuB__dzERat0XF9g2VtQgr9PJbu3XOizj5RZmh7AAuHIm4Bh-0Qc_lF5YKt_08W2Fp5jujGbdS9uJdbF9CUAr7t1dnZcAcQjbKBYNX4BAynRFdiuB--f_nZLgrnbyTyWz075vRK5h6xBARLIARNPvkSjtQBMH1b1L07Qe7K0GarZRmB_eSN9383LcOLn6_d0--xi12jzDwusC-eOkHWESqtFZESc6BfI7noOPqvhJlphCnvWh6IeYI2w9QOYEUIpUTI8np6LbgGY9Fs98rqVt5AXLIhWkWywlvmtVrBp0igcN_IoypGlUPQGe77Rw" },
    { "protected": "eyJhbGciOiJFUzI1NiJ9",
      "header": { "kid": "e9bc097a-ce51-4036-9562-d2ade882db0d" },
      "signature": "DtEhU3ljBEG8L38VWafUAqOyKAM6-Xx-F4GawxaePMXFCgftTjDxw5djxLa8ISlSApmWQxfKTUJqPP3-Kg6NUlQ" } ]
}
```

Appendix B. "x5c" (X.509 Certificate Chain) Example

The JSON array below is an example of a certificate chain that could be used as the value of an "x5c" (X.509 Certificate Chain) Header Parameter, per Section 4.1.6. Note that since these strings contain base64 encoded (not base64url encoded) values, they are allowed to contain white space and line breaks.

```
[ "MIIE3jCCA8agAwIBAgICAwEwDQYJKoZIhvcNAQEFBQAwYzELMAkGA1UEBhMCVVMxITAfBgNVBAoTGFROZSBHbyBEYWRkeSBHcm91cCwgSW5jLjExMC8GA1UECxMoR28gRGFkZHZkgQ2xhc3MgMiBDZXJ0aWZpY2F0aW9uIEF1dGhvcml0eTAeFw0wNjExMTYwMTU0MzdaFw0yNjExMTYwMTU0MzdaMlHKMQswCQYDVQGEWJVUzEQMA4GA1UECBMHQXJpem9uYTETMBEGA1UEBxMKU2NvdHRzZGFsZTEaMBGGA1UEChMRR29EYWRkeS5jb20sIEluYy4xMzAxBgNVBAsTKmh0dHA6Ly9jZXJ0aWZpY2F0ZXMuZ29kYWwRkeS5jb20vcmlvbmVwb3NpdG9yeTEwMC4GA1UEAxMnR28gRGFkZHZkgU2VjdXJlIENlc nRpZmljYXRpb24gQXV0aG9yaXR5MREwDwYDVQQFEWgwNzk2OTI4NzCCASIdQYJKoZIhvcNAQEBBQADggEPADCCAQoCggEBAMQt1RWMnCZM7DI161+4WQFapmGBWTTwY6vj3D3HkrjJM9N55DrtPDajhI6zMBS2sofDPZVUBJ7fmd0LJR4h3mUpfjWqVTr9vcyOdQmVZWt7/v+WibXnvQAJYwqDL1CBM6nPWT27oDyqu9SoWlm2r4arV3aL GbqGmu75RpRSgAvSMEYddi5Kcju+GZtCpyz8/x4fKL4o/Klw/O5epHBp+YlLpyo7RJlbmr2EkRTcDCVw5wrWCs9CHRK8r5RsL+H0EwnWGu1NcWdrxcx+AuP7q2BNgWJCJjPOq8lh8BJ6qf9Z/dFjpfMFDniNoW1fho3/Rb2cRGadDAW/hOUoz+EDU8CAW EAAaOCATiWggEuMB0GA1UdDgQWBBT9rGEYk2xFluLuhV+auud2mWjm5zAfBgNVHSMEGDAWgBTSxLDSkdrMEXGzYcs9of7dqGrU4zASBgNVHRMBAf8ECDAGAQH/AgEAMDMGCCsGAQUFBwEBBCCwJTAjBggrBgEFBQcwAYYXaHR0cDovL29jc3AuZ29kYWwR
```

keS5jb20wRgYDVR0fBD8wPTA7oDmgN4Y1aHR0cDovL2N1cnRpZmljYXRlc3Y5nb2RhZGR5LmNvbS9yZXBvc2l0b3J5L2dkcm9vdC5jcmwwSwYDVR0gBEQwQjBAbGRVHSAAMDgwNgYIKwYBBQUHAgEWMh0dHA6Ly9jZXJ0aWZpY2F0ZXMuZ29kYWRkeS5jb20vcmlvbmVwb3NpdG9yeTAOBgNVHQ8BAf8EBAMCAQYwDQYJKoZIhvcNAQEFBQADggEBANKGwOy9+aG2Z+5mC6IGOG9RQjhVyrEp0lVPLN8tESe8HkGsz2ZbwlFaleZAFPIUyIXvJxwqoJKSQ3kbTJSMUA2fCENZvD117esyfxVgqwcSeIaha86ykRvOe5GPLL5CkKSkB2XIsKd83ASe8T+5o0yGPwLPk9Qnt0hCqU7S+8MxZC9Y7lhyVJEnfzuz9p0iRFEU00jZv2kWzRaJBydTXRE4+uXR21aITVSzGh60lmawGhId/dQb8vxRMDsxuxN89txJx90jxUUAiKEngHUuHqDTMBqLdElrRhjZkAzVvb3du6/KFUJheqwnTrZEjYx8WnM25sgVjOuh0aBsXBTWVU+4=" ,

"MIIE+ZCCBGSGAwIBAgICAQ0wDQYJKoZIhvcNAQEFBQAwwbsxJDAiBgNVBACGTG1ZhbG1DZXJ0IFZhbkYXRpb24gTmV0d29yazEXMBUGA1UEChMOVmfSaUNlcnQsIEluYy4xNTAzBgNVBAsTLFZhbk1DZXJ0IENsYXNzIDIGUG9saWN5IFZhbkYXRpb24gQXV0aG9yaXR5MSEwHwYDVQQDEExodHRwOi8vd3d3LnZhbk1DZXJ0LmNvbS8xIDAeBgkqhkiG9w0BCQEWEluZm9AdmFsaWNlcnQuY29tMB4XDTA0MDYyOTE3MDYyMfOxDTI0MDYyOTE3MDYyMfOwYzELMAKGA1UEBhMCVVMxITAFBgNVBAoTGFROZSBBhbyBEYWRkeSBHcm91cCwgSW5jLjExMC8GA1UECXMOR28gRGFkZkZkZkQ2xhc3MgM1BDZXJ0aWZpY2F0aW9uIEF1dGhvcml0eTCCASAwDQYJKoZIhvcNAQEFBQADggENADCCAQgCggEBAN6dl+pXGEmhW+vXX0iG6r7d/+TvZxz0ZWizV3GgXne77ZtJ6XCAPVYYYwhv2vLM0D9/AlQiVBDYsoHUWU9S3/Hd8M+eKsa7Ugay9qK7HFih7Eux6wwdhFJ2+qN1j3hybX2C32qRe3H3I2TqYXP2WYktsqbl2i/ojgC95/5Y0V4evLOtXiEqITLdiOr18SPaAIBQi2XKVLOARFmR6jYGB0xUGlcmIbYsUfbl8aQr4CUWwoRiMYavx4A61nf4DD+qta/KFApMoZfV6yyO9ecw3ud72a9nmYvLEHZ6IVDD2gWMWZewo+YihfukEHU1jPEX44dMX4/7VpkI+EdOqXG68CAQ0jggHhMIIB3TAdBgNVHQ4EFgQU0sSw0pHUTBFxs2HLPaH+3ahq1OMwgdIGA1UdIwSByjCBx6GBwaSBvjCBuzEkMCIGA1UEBxMbVmfSaUNlcnQgVmFsaWRhdGlvbiBOZXR3b3JrMRcwFQYDQgQKEw5WYwXpQ2VydCwgSW5jLjE1MDMGA1UECXMsVmFsaUNlcnQgQ2xhc3MgMiBQb2xpyY3kgVmFsaWRhdGlvbiBBdXR0b3JpdHkxITAFBgNVBAMTGh0dHA6Ly93d3ducmlvbmVwb3NpdG9yeTAOBgNVHQ8BAf8EBAMCAQYwDQYJKoZIhvcNAQEFBQADggEBANKGwOy9+aG2Z+5mC6IGOG9RQjhVyrEp0lVPLN8tESe8HkGsz2ZbwlFaleZAFPIUyIXvJxwqoJKSQ3kbTJSMUA2fCENZvD117esyfxVgqwcSeIaha86ykRvOe5GPLL5CkKSkB2XIsKd83ASe8T+5o0yGPwLPk9Qnt0hCqU7S+8MxZC9Y7lhyVJEnfzuz9p0iRFEU00jZv2kWzRaJBydTXRE4+uXR21aITVSzGh60lmawGhId/dQb8vxRMDsxuxN89txJx90jxUUAiKEngHUuHqDTMBqLdElrRhjZkAzVvb3du6/KFUJheqwnTrZEjYx8WnM25sgVjOuh0aBsXBTWVU+4=" ,

"MIIC5zCCALACAQEWdQYJKoZIhvcNAQEFBQAwwbsxJDAiBgNVBACGTG1ZhbG1DZXJ0IFZhbkYXRpb24gTmV0d29yazEXMBUGA1UEChMOVmfSaUNlcnQsIEluYy4xNTAzBgNVBAsTLFZhbk1DZXJ0IENsYXNzIDIGUG9saWN5IFZhbkYXRpb24gQXV0aG9yaXR5MSEwHwYDVQQDEExodHRwOi8vd3d3LnZhbk1DZXJ0LmNvbS8xIDAeBgkqhkiG9w0BCQEWEluZm9AdmFsaWNlcnQuY29tMB4XDTA0MDYyOTE3MDYyMfOxDTI0MDYyOTE3MDYyMfOwYzELMAKGA1UEBhMCVVMxITAFBgNVBAoTGFROZSBBhbyBEYWRkeSBHcm91cCwgSW5jLjExMC8GA1UECXMOR28gRGFkZkZkZkQ2xhc3MgM1BDZXJ0aWZpY2F0aW9uIEF1dGhvcml0eTCCASAwDQYJKoZIhvcNAQEFBQADggENADCCAQgCggEBAN6dl+pXGEmhW+vXX0iG6r7d/+TvZxz0ZWizV3GgXne77ZtJ6XCAPVYYYwhv2vLM0D9/AlQiVBDYsoHUWU9S3/Hd8M+eKsa7Ugay9qK7HFih7Eux6wwdhFJ2+qN1j3hybX2C32qRe3H3I2TqYXP2WYktsqbl2i/ojgC95/5Y0V4evLOtXiEqITLdiOr18SPaAIBQi2XKVLOARFmR6jYGB0xUGlcmIbYsUfbl8aQr4CUWwoRiMYavx4A61nf4DD+qta/KFApMoZfV6yyO9ecw3ud72a9nmYvLEHZ6IVDD2gWMWZewo+YihfukEHU1jPEX44dMX4/7VpkI+EdOqXG68CAQ0jggHhMIIB3TAdBgNVHQ4EFgQU0sSw0pHUTBFxs2HLPaH+3ahq1OMwgdIGA1UdIwSByjCBx6GBwaSBvjCBuzEkMCIGA1UEBxMbVmfSaUNlcnQgVmFsaWRhdGlvbiBOZXR3b3JrMRcwFQYDQgQKEw5WYwXpQ2VydCwgSW5jLjE1MDMGA1UECXMsVmFsaUNlcnQgQ2xhc3MgMiBQb2xpyY3kgVmFsaWRhdGlvbiBBdXR0b3JpdHkxITAFBgNVBAMTGh0dHA6Ly93d3ducmlvbmVwb3NpdG9yeTAOBgNVHQ8BAf8EBAMCAQYwDQYJKoZIhvcNAQEFBQADggEBANKGwOy9+aG2Z+5mC6IGOG9RQjhVyrEp0lVPLN8tESe8HkGsz2ZbwlFaleZAFPIUyIXvJxwqoJKSQ3kbTJSMUA2fCENZvD117esyfxVgqwcSeIaha86ykRvOe5GPLL5CkKSkB2XIsKd83ASe8T+5o0yGPwLPk9Qnt0hCqU7S+8MxZC9Y7lhyVJEnfzuz9p0iRFEU00jZv2kWzRaJBydTXRE4+uXR21aITVSzGh60lmawGhId/dQb8vxRMDsxuxN89txJx90jxUUAiKEngHUuHqDTMBqLdElrRhjZkAzVvb3du6/KFUJheqwnTrZEjYx8WnM25sgVjOuh0aBsXBTWVU+4=" ,

"MIIC5zCCALACAQEWdQYJKoZIhvcNAQEFBQAwwbsxJDAiBgNVBACGTG1ZhbG1DZXJ0IFZhbkYXRpb24gTmV0d29yazEXMBUGA1UEChMOVmfSaUNlcnQsIEluYy4xNTAzBgNVBAsTLFZhbk1DZXJ0IENsYXNzIDIGUG9saWN5IFZhbkYXRpb24gQXV0aG9yaXR5MSEwHwYDVQQDEExodHRwOi8vd3d3LnZhbk1DZXJ0LmNvbS8xIDAeBgkqhkiG9w0BCQEWEluZm9AdmFsaWNlcnQuY29tMIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQDOOnHK5a

```
vIWZJV16vYdA757tn2VUDZZUcOBVXc65g2PFxTXdMwzzjsvUGJ7SVCCSRrCl6zf
N1SLUzm1NZ9WlmpZdRJEy0kTRxQb7XBhVQ7/nHk01xC+YDgkRoKWzk2Z/M/VXwb
P7RfZHM047QSV4dk+NoS/zcnwbNDu+97bi5p9wIDAQABMA0GCSqGSIB3DQEBBQU
AA4GBADt/UG9vUJSZSWI4OB9L+KXIPqeCgfYrx+jFzug6EILLGACOTb2oWH+heQ
Clu+mNr0HZDzTuIYEZoDJJKPTEjlbVUjP9UNV+mWwD5MlM/Mtsq2azSiGM5bUMM
j4QssxsodyamEwCW/POuZ6lcg5Ktz885hZo+L7tdEy8W9ViH0Pd"]
```

Appendix C. Notes on implementing base64url encoding without padding

This appendix describes how to implement base64url encoding and decoding functions without padding based upon standard base64 encoding and decoding functions that do use padding.

To be concrete, example C# code implementing these functions is shown below. Similar code could be used in other languages.

```
static string base64urlencode(byte [] arg)
{
    string s = Convert.ToBase64String(arg); // Regular base64 encoder
    s = s.Split('=')[0]; // Remove any trailing '='s
    s = s.Replace('+', '-'); // 62nd char of encoding
    s = s.Replace('/', '_'); // 63rd char of encoding
    return s;
}

static byte [] base64urldecode(string arg)
{
    string s = arg;
    s = s.Replace('-', '+'); // 62nd char of encoding
    s = s.Replace('_', '/'); // 63rd char of encoding
    switch (s.Length % 4) // Pad with trailing '='s
    {
        case 0: break; // No pad chars in this case
        case 2: s += "=="; break; // Two pad chars
        case 3: s += "="; break; // One pad char
        default: throw new System.Exception(
            "Illegal base64url string!");
    }
    return Convert.FromBase64String(s); // Standard base64 decoder
}
```

As per the example code above, the number of '=' padding characters that needs to be added to the end of a base64url encoded string without padding to turn it into one with padding is a deterministic function of the length of the encoded string. Specifically, if the length mod 4 is 0, no padding is added; if the length mod 4 is 2, two '=' padding characters are added; if the length mod 4 is 3, one '='

padding character is added; if the length mod 4 is 1, the input is malformed.

An example correspondence between unencoded and encoded values follows. The octet sequence below encodes into the string below, which when decoded, reproduces the octet sequence.

3 236 255 224 193

A-z_4ME

Appendix D. Negative Test Case for "crit" Header Parameter

Conforming implementations must reject input containing critical extensions that are not understood or cannot be processed. The following JWS must be rejected by all implementations, because it uses an extension Header Parameter name "http://example.invalid/UNDEFINED" that they do not understand. Any other similar input, in which the use of the value "http://example.invalid/UNDEFINED" is substituted for any other Header Parameter name not understood by the implementation, must also be rejected.

The JWS Protected Header value for this JWS is:

```
{ "alg": "none",  
  "crit": [ "http://example.invalid/UNDEFINED" ],  
  "http://example.invalid/UNDEFINED": true  
}
```

The complete JWS that must be rejected is as follows (with line breaks for display purposes only):

```
eyJhbGciOiJub25lIiwNCiAiY3JpdCI6WyJodHRwOi8vZXhhbXBsZS5jb20vVU5ERU  
ZJTkVEIl0sDQogImh0dHA6Ly9leGFtcGxlLnNvbS9VTkRFRklORUQiOnRydWUNCn0.  
RkFJTJA.
```

Appendix E. Acknowledgements

Solutions for signing JSON content were previously explored by Magic Signatures [MagicSignatures], JSON Simple Sign [JSS], and Canvas Applications [CanvasApp], all of which influenced this draft.

Thanks to Axel Nennker for his early implementation and feedback on the JWS and JWE specifications.

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, Brian Campbell, Breno de Medeiros, Dick Hardt, Joe Hildebrand, Jeff Hodges, Edmund Jay, Yaron Y. Goland, Ben Laurie, James Manger, Matt Miller, Tony Nadalin, Axel Nennker, John Panzer, Emmanuel Raviart, Eric Rescorla, Jim Schaad, Paul Tarjan, 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 F. Document History

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

-17

- o Refined the "typ" and "cty" definitions to always be MIME Media Types, with the omission of "application/" prefixes recommended for brevity, addressing issue #50.
- o Updated the mandatory-to-implement (MTI) language to say that general-purpose implementations must implement the single signature/MAC value case for both serializations whereas special-purpose implementations can implement just one serialization if that meets the needs of the use cases the implementation is designed for, addressing issue #119.
- o Explicitly named all the logical components of a JWS and defined the processing rules and serializations in terms of those components, addressing issues #60, #61, and #62.
- o Replaced verbose repetitive phrases such as "base64url encode the octets of the UTF-8 representation of X" with mathematical notation such as "BASE64URL(UTF8(X))".
- o 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.

-16

- o Changes to address editorial and minor issues #50, #98, #99, #102, #104, #106, #107, #111, and #112.

-15

- o Clarified that it is an application decision which signatures, MACs, or plaintext values must successfully validate for the JWS to be accepted, addressing issue #35.
- o Corrected editorial error in "ES512" example.
- o Changes to address editorial and minor issues #34, #96, #100, #101, #104, #105, and #106.

-14

- o Stated that the "signature" parameter is to be omitted in the JWS JSON Serialization when its value would be empty (which is only the case for a Plaintext JWS).

-13

- o Made all header parameter values be per-signature/MAC, addressing issue #24.

-12

- o Clarified that the "typ" and "cty" header parameters are used in an application-specific manner and have no effect upon the JWS processing.
- o Replaced the MIME types "application/jws+json" and "application/jws" with "application/jose+json" and "application/jose".
- o Stated that recipients MUST either reject JWSs with duplicate Header Parameter Names or use a JSON parser that returns only the lexically last duplicate member name.
- o Added a Serializations section with parallel treatment of the JWS Compact Serialization and the JWS JSON Serialization and also moved the former Implementation Considerations content there.

-11

- o Added Key Identification section.
- o For the JWS JSON Serialization, enable header parameter values to be specified in any of three parameters: the "protected" member that is integrity protected and shared among all recipients, the "unprotected" member that is not integrity protected and shared

among all recipients, and the "header" member that is not integrity protected and specific to a particular recipient. (This does not affect the JWS Compact Serialization, in which all header parameter values are in a single integrity protected JWE Header value.)

- o Removed suggested compact serialization for multiple digital signatures and/or MACs.
- o Changed the MIME type name "application/jws-js" to "application/jws+json", addressing issue #22.
- o Tightened the description of the "crit" (critical) header parameter.
- o Added a negative test case for the "crit" header parameter

-10

- o Added an appendix suggesting a possible compact serialization for JWSs with multiple digital signatures and/or MACs.

-09

- o Added JWS JSON Serialization, as specified by draft-jones-jose-jws-json-serialization-04.
- o Registered "application/jws-js" MIME type and "JWS-JS" typ header parameter value.
- o Defined that the default action for header parameters that are not understood is to ignore them unless specifically designated as "MUST be understood" or included in the new "crit" (critical) header parameter list. This addressed issue #6.
- o Changed term "JWS Secured Input" to "JWS Signing Input".
- o Changed from using the term "byte" to "octet" when referring to 8 bit values.
- o Changed member name from "recipients" to "signatures" in the JWS JSON Serialization.
- o Added complete values using the JWS Compact Serialization for all examples.

-08

- o Applied editorial improvements suggested by Jeff Hodges and Hannes Tschofenig. Many of these simplified the terminology used.
- o Clarified statements of the form "This header parameter is OPTIONAL" to "Use of this header parameter is OPTIONAL".
- o Added a Header Parameter Usage Location(s) field to the IANA JSON Web Signature and Encryption Header Parameters registry.
- o Added seriesInfo information to Internet Draft references.

-07

- o Updated references.

-06

- o Changed "x5c" (X.509 Certificate Chain) representation from being a single string to being an array of strings, each containing a single base64 encoded DER certificate value, representing elements of the certificate chain.
- o Applied changes made by the RFC Editor to RFC 6749's registry language to this specification.

-05

- o Added statement that "StringOrURI values are compared as case-sensitive strings with no transformations or canonicalizations applied".
- o Indented artwork elements to better distinguish them from the body text.

-04

- o Completed JSON Security Considerations section, including considerations about rejecting input with duplicate member names.
- o Completed security considerations on the use of a SHA-1 hash when computing "x5t" (x.509 certificate thumbprint) values.
- o Refer to the registries as the primary sources of defined values and then secondarily reference the sections defining the initial contents of the registries.
- o Normatively reference XML DSIG 2.0 [W3C.CR-xmlsig-core2-20120124] for its security considerations.

- o Added this language to Registration Templates: "This name is case sensitive. Names that match other registered names in a case insensitive manner SHOULD NOT be accepted."
- o Reference draft-jones-jose-jws-json-serialization instead of draft-jones-json-web-signature-json-serialization.
- o Described additional open issues.
- o Applied editorial suggestions.

-03

- o Added the "cty" (content type) header parameter for declaring type information about the secured content, as opposed to the "typ" (type) header parameter, which declares type information about this object.
- o Added "Collision Resistant Namespace" to the terminology section.
- o Reference ITU.X690.1994 for DER encoding.
- o Added an example JWS using ECDSA P-521 SHA-512. This has particular illustrative value because of the use of the 521 bit integers in the key and signature values. This is also an example in which the payload is not a base64url encoded JSON object.
- o Added an example "x5c" value.
- o No longer say "the UTF-8 representation of the JWS Secured Input (which is the same as the ASCII representation)". Just call it "the ASCII representation of the JWS Secured Input".
- o Added Registration Template sections for defined registries.
- o Added Registry Contents sections to populate registry values.
- o Changed name of the JSON Web Signature and Encryption "typ" Values registry to be the JSON Web Signature and Encryption Type Values registry, since it is used for more than just values of the "typ" parameter.
- o Moved registries JSON Web Signature and Encryption Header Parameters and JSON Web Signature and Encryption Type Values to the JWS specification.
- o Numerous editorial improvements.

-02

- o Clarified that it is an error when a "kid" value is included and no matching key is found.
- o Removed assumption that "kid" (key ID) can only refer to an asymmetric key.
- o Clarified that JWSs with duplicate Header Parameter Names MUST be rejected.
- o Clarified the relationship between "typ" header parameter values and MIME types.
- o Registered application/jws MIME type and "JWS" typ header parameter value.
- o Simplified JWK terminology to get replace the "JWK Key Object" and "JWK Container Object" terms with simply "JSON Web Key (JWK)" and "JSON Web Key Set (JWK Set)" and to eliminate potential confusion between single keys and sets of keys. As part of this change, the Header Parameter Name for a public key value was changed from "jpk" (JSON Public Key) to "jwk" (JSON Web Key).
- o Added suggestion on defining additional header parameters such as "x5t#S256" in the future for certificate thumbprints using hash algorithms other than SHA-1.
- o Specify RFC 2818 server identity validation, rather than RFC 6125 (paralleling the same decision in the OAuth specs).
- o Generalized language to refer to Message Authentication Codes (MACs) rather than Hash-based Message Authentication Codes (HMACs) unless in a context specific to HMAC algorithms.
- o Reformatted to give each header parameter its own section heading.

-01

- o Moved definition of Plaintext JWSs (using "alg":"none") here from the JWT specification since this functionality is likely to be useful in more contexts than just for JWTs.
- o Added "jpk" and "x5c" header parameters for including JWK public keys and X.509 certificate chains directly in the header.
- o Clarified that this specification is defining the JWS Compact Serialization. Referenced the new JWS-JS spec, which defines the

JWS JSON Serialization.

- o Added text "New header parameters should be introduced sparingly since an implementation that does not understand a parameter MUST reject the JWS".
- o Clarified that the order of the creation and validation steps is not significant in cases where there are no dependencies between the inputs and outputs of the steps.
- o Changed "no canonicalization is performed" to "no canonicalization need be performed".
- o Corrected the Magic Signatures reference.
- o Made other editorial improvements suggested by JOSE working group participants.

-00

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

Authors' Addresses

Michael B. Jones
Microsoft

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

John Bradley
Ping Identity

Email: ve7jtb@ve7jtb.com

Nat Sakimura
Nomura Research Institute

Email: n-sakimura@nri.co.jp

