

Network Working Group
Internet-Draft
Intended status: Standards Track
Expires: December 17, 2012

R. Barnes
BBN Technologies
June 15, 2012

JavaScript Message Security Format
draft-barnes-jose-jsms-00.txt

Abstract

Many applications require the ability to send cryptographically secured messages. While the IETF has defined a number of formats for such messages (e.g. CMS) those formats use encodings which are not easy to use in modern applications. This document describes the JavaScript Message Security format (JSMS), a new cryptographic message format which is based on JavaScript Object Notation (JSON) and thus is easy for many applications to generate and parse.

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 December 17, 2012.

Copyright Notice

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

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

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

Table of Contents

1. Introduction	3
2. Conventions Used In This Document	3
3. Overview	3
3.1. Operational Modes	4
3.2. Design Principles	4
3.3. Certificate Processing	5
3.4. Certificate Discovery	5
4. Message Format	5
4.1. Data types	6
4.2. Basic Types	6
4.3. SignedData	7
4.3.1. Signature	7
4.3.2. Generating a SignedData Object	8
4.3.3. Verifying a SignedData Object	8
4.4. AuthenticatedData	9
4.4.1. Generating an AuthenticatedData Object	9
4.4.2. Verifying an AuthenticatedData Object	9
4.5. EncryptedData	10
4.5.1. Generating an EncryptedData Object	10
4.5.2. Decrypting a EncryptedData Object	11
4.6. Useful Objects	11
4.6.1. AlgorithmIdentifier	11
4.6.2. PublicKey	14
4.6.3. WrappedKey	16
5. Compact Format	17
6. Examples	18
6.1. Parameters	18
6.2. SignedData	18
6.3. AuthenticatedData	19
6.4. EncryptedData	20
7. Mapping to CMS	20
8. Comparison to JWS/JWE/JWK	21
9. IANA Considerations	22
10. Security Considerations	22
11. Acknowledgements	23
12. References	23
12.1. Normative References	23
12.2. Informative References	24
Appendix A. Acknowledgments	25
Author's Address	25

1. Introduction

Many applications require the ability to send cryptographically secured (encrypted, digitally signed, etc.) messages. While the IETF has defined a number of formats for such messages, those formats are widely viewed as being excessively complicated for the demands of Web applications, which typically only need the ability to secure simple messages. In addition, existing formats use encoding mechanisms (e.g., ASN.1 DER) which are not congenial for many classes of applications (e.g., Web applications). This presents an obstacle to the deployment of strong security by such applications.

This document describes a new cryptographic message format, JavaScript Message Security (JSMS). This format is intended to meet the need of modern applications, including JavaScript-based Web applications. While JSMS is modeled on existing formats -- principally CMS [RFC5652] -- it uses JavaScript Object Notation (JSON) rather than ASN.1, making it far easier for applications to handle. In the interest of simplicity, JSMS also omits many of less commonly used CMS modes (such as password-based encryption).

2. Conventions Used In This Document

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 [RFC2119].

In order to enable JSON to carry binary data, JSMS makes extensive use of Base64 encoding [RFC4648]. Whenever this document refers to Base64 encoding, we mean the URL-safe variant "base64url" encoding. As stated in section 3.1 of [RFC4648], Base64 does not allow linefeeds. Since linefeeds are not valid characters in a JSON string, whenever a field is specified to be Base64-encoded in this document, it MUST NOT include any line breaks. Base64-encoded fields also MUST NOT include JSON-encoded linefeeds such as "\n". Any linebreaks in the middle of Base64-encoded sections of the examples in this document have been inserted in order to make the examples fit on the page. Any trailing "=" characters SHOULD be removed. They are not needed, because JSON strings have defined lengths (namely the number of characters between unescaped '"' characters).

3. Overview

The JSMS message format is simply a JSON [RFC4627] object with an appropriate collection of fields. Each operating mode will have a separate set of fields, with a common field to distinguish between

the modes.

3.1. Operational Modes

JSMS supports three operational modes:

Signed Data

A block of data signed by a single signer using his asymmetric key and optionally carrying his certificate.

Authenticated Data

A block of data with authentication and integrity protection provided using a symmetric-key Message Authentication Code (MAC). The MAC key may be provided in encrypted form (as with Encrypted Data) or identified by name.

Encrypted Data

A block of data encrypted under a random message encryption key (MEK). The MEK is then separately encrypted for each recipient, either via symmetric or asymmetric encryption. The data is always integrity protected, through the use of an Authenticated Encryption with Associated Data (AEAD) algorithm such as AES-GCM or AES-CCM.

Any other desired security functions are provided by composition of these modes. For instance, a signed and encrypted message is produced by first creating a Signed message and then encrypting that data.

3.2. Design Principles

In general, JSMS follows the following design principles.

Minimize implementation complexity

Wherever possible, protocol choices have been made such that the time and effort required to implement the protocol in many different programming languages will be minimized. This means that optimizations for bandwidth, CPU, and memory utilization have been explicitly avoided.

Base64 as the only encoding

Any data that does not have a straightforward string representation (binary values, large integers, etc.) is base64-encoded (see: [RFC4648]). In some cases, hexadecimal encodings might be more convenient, but consistency is even more important to reduce implementation complexity.

No canonicalization

In many cryptographic message formats, canonical encodings are used to allow the same value to be computed at both sender and recipient (e.g., for digital signatures). This is inconvenient in JSON, which just views messages as a bundle of key/value pairs. Instead, whenever canonicalization would be required, the relevant

data is serialized and base64-encoded for transport, allowing both sides to run computations over the same original set of octets.

In-memory processing

We assume that the entire message can fit in main memory and make no effort to design a wire representation which can be handled in small chunks in a single pass. This means, for instance, that there is no need to have a message digest indicator at the beginning of the message and then the signature at the end, as is done in CMS. Fields are simply serialized in whatever order is most convenient for the JSON implementation. The examples in this document are generally shown in whatever order seems most readable and are not normative.

Consistency with CMS

To simplify the adaptation of existing cryptographic modules and the validation of JSMS implementations, changes from the CMS cryptographic operations are minimized. JSMS is semantically equivalent to a profile of CMS, as described in Section 7.

3.3. Certificate Processing

Experience has shown that certificate handling (path construction) is one of the trickier parts of building a cryptographic system. While JSMS supports PKIX certificates, its certificate processing is far simpler than that of CMS. (It also supports the use of bare public keys in order to avoid the use of X.509 altogether.) When a JSMS agent provides its certificate, it must provide an ordered chain (as in TLS [RFC5246]) terminating in its own certificate, thus removing the need to construct certificate paths. The certificates MUST be ordered with the end-entity certificate first and each certificate that follows signing the certificate immediately preceding it.

3.4. Certificate Discovery

JSMS will often be used in an online messaging environments with users that have an address of the form user@domain, such as email, XMPP, or SIP. As such, protocols such as WebFinger [I-D.hammer-webfinger] or an end-to-end protocol can be used to retrieve appropriate certificates. Downstream uses of JSMS SHOULD define a discovery mechanism suitable for the intended use.

4. Message Format

A JSMS object is a JSON object that encodes cryptographic information related to a content byte string. This document specifies the set of keys that must be present in a JSMS object, what the associated values are, and how these values are generated and processed in order to realize security features. In processing JSMS objects, unknown

keys MUST be ignored.

JSMS defines three top-level types of secure object, each of which provides a specific cryptographic protection to a byte string.

SignedData: Signature using a public-key digital signature algorithm

AuthenticatedData: Authentication using a Message Authentication Code (MAC)

EncryptedData: Encryption and authentication using an Authenticated Encryption with Associated Data (AEAD) algorithm

4.1. Data types

For each field in a JSON object, we define the type of information that must be included in that field. At base are the object, array, string, number types defined by JSON. We also use two special subclasses of strings: Fields with type "Token" contain a string drawn from a defined list of strings (e.g., an IANA registry for algorithm names). Fields with type "ByteString" contain a Base64-encoded byte string (note the considerations related to Base64 encoding in Section 2 above).

In addition to the primitive data types, Section 4.6 defines a collection of useful object types that are used by the top-level JSMS objects. These are simply referred to by name when they appear as a field value in another object.

4.2. Basic Types

The following elements are common to all JSMS messages:

"version": REQUIRED Number. The version of JSMS used by this object. This field MUST be set to 1.
"type": REQUIRED Token. The type of this JSMS object. This field MUST be set to one of the following values
 "signed": SignedData object
 "authenticated": AuthenticatedData object
 "encrypted": EncryptedData object
"content": OPTIONAL ByteString. The content byte string, Base64-encoded.

If the "content" key is not present in a given JSMS object, then the JSMS object is "detached". In this case, the content must be associated with the JSMS object through some out-of-band mechanism before the JSMS object can be processed. Note that there is a risk that detached JSMS object might become invalid if the content is transformed, even if this transformation preserves the semantics of the content. For example, if the content is a JSON object, and the

object passes through an intermediate process that adds whitespace or re-orders the fields in the object (neither of which changes the meaning of the object), then the recipient will not be able to verify the signature. For this reason, detached JSMS objects SHOULD NOT be used unless there is a canonical form for the content being processed.

4.3. SignedData

A SignedData object MUST have a "type" field set to "signed". In addition, a SignedData object contains the following keys:

- "digestAlgorithm": REQUIRED AlgorithmIdentifier. The digest algorithm used in signing the content.
- "signatures": REQUIRED Array of Signature. One or more digital signatures over the content.
- "certificates": OPTIONAL Array of String. A certificate chain associating the signer's public key with an identifier. Each element in the array is a string containing the Base64-encoded representation of a DER-formatted certificate. The certificates MUST be ordered with the end-entity certificate first and each certificate that follows signing the certificate immediately preceding it.
- "certificatesURI": OPTIONAL String. An HTTP or HTTPS URI referring to a certificate chain. The referenced resource MUST have type "application/json" and contain an array of certificates in the same format as the "certificates" element above, including the ordering constraint.

4.3.1. Signature

A Signature object represents the signature over the content in the SignedData object by a specific key pair. A Signature object can contain the following keys:

- "signatureAlgorithm": REQUIRED AlgorithmIdentifier. The signature algorithm used in signing the content.
- "key": REQUIRED PublicKey. The public key identifier for the signer, represented as a PublicKey object (see Section 4.6.2)
- "signature": REQUIRED ByteString. The Base64-encoded signature value

If the "key" value represents the public key as an identifier, then a certificate for the signer MUST be provided by setting either the "certificates" or "certificatesURL" fields. The subject key in the end-entity certificate MUST match the identifier in the "key" value; the certificate SHOULD contain the subjectKeyIdentifier field, with a value matching the "key" value. (Note that this implies that when there are multiple signers, only one key can be represented by ID.)

4.3.2. Generating a SignedData Object

The inputs to the process of generating a SignedData object are:

- o The content, as a byte string
- o A digest algorithm
- o One or more signature algorithms and asymmetric key pairs

To generate the signature for SignedData object, the originator takes the following steps:

1. Compute the message digest by applying the digest algorithm to the content.
2. For each signing key pair, compute the signature by using the signature algorithm to sign the message digest with the private key from the asymmetric key pair.

The originator then encodes the SignedData object by including the appropriate AlgorithmIdentifiers for the digest algorithms, a Signature object for each signature, and (optionally) the content.

4.3.3. Verifying a SignedData Object

To verify a SignedData object, the recipient takes the following steps:

1. Verify that the digest and signature algorithms are supported. Otherwise, report an error and fail.
2. Compute the content byte string by decoding the "content" value of the JSMS object. If the JSMS object does not contain a "content" field, retrieve the content by other means.
3. Compute the message digest by applying the digest algorithm to the content.
4. Compute the signature by decoding the "signature" value of the JSMS object.
5. Compute the public key:
 - * If the key is represented directly, then decode it according to the rules specified by the algorithm name.
 - * If the key is represented by an ID, then retrieve the corresponding subject public key from the end-entity certificate. If no "certificates" or "certificatesURI" value is present, then report an error and fail.
 - * If the key is represented by a URI, retrieve the public key from the URI.
6. Verify the signature by using the signature algorithm to verify the message digest with the public key.

4.4. AuthenticatedData

An AuthenticatedData object MUST have a "type" field set to "authenticated". In addition, an AuthenticatedData object contains the following keys:

"algorithm": REQUIRED AlgorithmIdentifier. The MAC algorithm used to authenticate the content.
"mac": REQUIRED ByteString. The MAC value
"keys": OPTIONAL Array of WrappedKey. Wrapped versions of the symmetric key used for this MAC. Each element in the array MUST be a WrappedKey object (see below)
"keyId": OPTIONAL ByteString. An opaque identifier for a pre-shared MAC key

An AuthenticatedData object MUST contain either the "key" field or the "keyId" field, so that the recipient knows which key to use to verify the MAC.

4.4.1. Generating an AuthenticatedData Object

The inputs to the process of generating a AuthenticatedData object are:

- o The content, as a byte string
- o A MAC algorithm
- o A MAC key and key identifier, or
- o One or more recipient keys and key encipherment algorithms

If the recipient key is specified rather than the MAC key directly, then a random MAC key is generated and encoded in a WrappedKey objects for each recipient (see Section 4.6.3). Once the MAC key has been determined, the originator uses the MAC algorithm and MAC key to compute the MAC over the content byte string.

The originator then encodes the AuthenticatedData object by including the appropriate AlgorithmIdentifier for the MAC algorithm and the Base64 representations of the MAC value and (optionally) the content. If the MAC key was specified directly, then the Base64 representation of the key identifier is set as the "keyId" value; otherwise, the WrappedKey objects are collected in an array and set as the "keys" value.

4.4.2. Verifying an AuthenticatedData Object

To verify a AuthenticatedData object, the recipient takes the following steps:

1. Verify that the MAC algorithm is supported. If not, report an error and fail.
2. Compute the content byte string by decoding the "content" value of the JSMS object. If the JSMS object does not contain a "content" field, retrieve the content by other means.
3. Compute the MAC key:
 - * If the "keyId" value is present and represents a known key, use the identified key.
 - * If the "keys" value is present, check each WrappedKey object to determine if it matches a known key for this recipient. If any of the wrapped keys matches, unwrap the key from the first one and use it (see Section Section 4.6.3). Otherwise, report an error and fail.
4. Use the MAC algorithm and MAC key to compute the MAC over the content byte string
5. Decode the MAC value from the "mac" field.
6. Verify that the computed MAC matches the MAC from the object.

4.5. EncryptedData

An EncryptedData object MUST have a "type" field set to "encrypted". Note also that in an EncryptedData object, the "content" field contains the encrypted form of the content, not the content itself (as plaintext). An EncryptedData object contains the following keys in addition to any common fields:

"algorithm": REQUIRED AlgorithmIdentifier. The encryption algorithm used to encrypt the content

"keys": REQUIRED Array of WrappedKey. Wrapped versions of the symmetric key used to encrypt the content. Each element in the array MUST be a WrappedKey object (see Section 4.6.3).

"mac": OPTIONAL ByteString. The MAC value, if required by the algorithm

Note that although the "mac" field is optional, an EncryptedData object always has an integrity check. All of the encryption algorithms used in JSMS are "Authenticated Encryption with Associated Data" algorithms, which include an authentication / integrity function by definition. The MAC field is optional because some AEAD algorithms have a separate MAC value (e.g., GCM), while others incorporate the MAC value into the ciphertext (e.g., CCM).

4.5.1. Generating an EncryptedData Object

The inputs to the process of generating a SignedData object are:

- o The content, as a byte string
- o An encryption algorithm
- o One or more recipient keys and key encipherment algorithms

The originator generates a random encryption key of a length suitable for the encryption algorithm, then encodes it in a `WrappedKey` object for each recipient (see Section 4.6.3). The content is then encrypted using the generated encryption key and the specified encryption algorithm.

The originator then encodes the `EncryptedData` object by including the appropriate `AlgorithmIdentifier` for the encryption algorithm, an array containing the `WrappedKey` objects, and (optionally) the Base64 representation of the content.

4.5.2. Decrypting a `EncryptedData` Object

To decrypt an `EncryptedData` object, the recipient takes the following steps:

1. Verify that the encryption algorithm is supported. If not, report an error and fail.
2. Compute the content byte string by decoding the "content" value of the JSMS object. If the JSMS object does not contain a "content" field, retrieve the content by other means.
3. Locate the encryption key: Check each `WrappedKey` object to determine if it matches a known key for this recipient. If any of the wrapped keys matches, unwrap the key from the first one and use it (see Section 4.6.3). Otherwise, report an error and fail.
4. Decrypt the content using the encryption key and the specified encryption algorithm.
5. Verify that the integrity check in the AEAD decryption was successful. If not, report an error and fail.
6. Return the decrypted content.

4.6. Useful Objects

In this section we define some common object types that are used across the top-level objects above.

4.6.1. `AlgorithmIdentifier`

An `AlgorithmIdentifier` object names a cryptographic algorithm and specifies any associated parameters such as nonces or initialization vectors (IVs). If the algorithm has no parameters, then the `AlgorithmIdentifier` object is simply a token representing the name of the algorithm, drawn from an IANA registry of algorithm names.

If the algorithm specifies parameters, the AlgorithmIdentifier object is a JSON object. There is only one required field, "name". Any other fields are specified in the algorithm definition.

"name": REQUIRED Token. The name of the algorithm, chosen from one of the IANA registries defined by this document.

The following table summarizes the algorithms to be used with JSMS.
[[More detail to be added later, in a separate document]]

Name	Parameters	Reference	Example
=====			
SIGNING			
rsa	no	[RFC3447]	"rsa"
dsa	yes (p,q,g)	[FIPS186]	{name:"dsa", p:1, q:2, g:3}
ecdsa	yes (curve)	[RFC6090]	{name:"ecdsa", curve:"P-256"}

DIGEST			
sha1	no	[FIPS180-1]	"sha1"
sha256	no	[FIPS180-3]	"sha256"
sha384	no	[FIPS180-3]	"sha384"
sha512	no	[FIPS180-3]	"sha512"

MAC			
hs1	no	[FIPS180-1]	"hs1"
hs256	no	[FIPS180-3]	"hs256"
hs384	no	[FIPS180-3]	"hs384"
hs512	no	[FIPS180-3]	"hs512"

ENCRYPTION			
aes128-ccm	yes (n,M)	[RFC3610]	{name:"aes128-ccm", n:"ZONce...lU-g", m:8}
aes128-gcm	yes (iv)	[McGrew & Viega]	{name:"aes128-gcm", iv:"ZONce...lU-g"}

KEY ENCIPHERMENT			
aes	no	[RFC3394]	"aes"
rsaes-oaep	no	[RFC3447]	"rsaes-oaep"

KEY AGREEMENT			
dh-es	yes (group)	[RFC2631]	{name:"dh-es", group: 14}
ecdh-es	yes (curve)	[RFC6090]	{name:"ecdh-es", curve:"P-256"}
=====			

Obviously, there will be more detail needed beyond the above, and some IANA considerations to create the necessary registries. For some algorithms, there will be specific notes about how they are to

be used with JSMS, for example:

- o The signature value produced by DSA is comprised of two integers. The byte string to be filled in the "signature" field is the two-element JSON array containing two integers, "[r,s]"
- o RSAES-OAEP is always used with SHA-256 and the default MGF1 masking generation function
- o Elliptic curves may only be specified by name, not by directly specifying curve parameters. [[We may define our own registry, or re-use the ones from TLS/IKE.]]
- o AEAD algorithms are only used for authenticated encryption; there is never associated data. Further AEAD algorithms may be defined using [draft-mcgrew-aead-aes-cbc-hmac-sha1]

4.6.2. PublicKey

A PublicKey object describes the public key used by a signer. The key may be specified as a JSON structure directly, as a URI, or as an identifier. A PublicKey object has the following fields:

"type" OPTIONAL Token. The name of the algorithm with which this key is to be used
"id" OPTIONAL ByteString. An identifier for the key
"uri" OPTIONAL String. A URI pointing to a direct form of the key

If the key is specified directly, then the "type" key MUST be present; the "id" and "uri" fields MAY be present. Subsequent entries in the array specify the elements of the key, in a manner determined by the algorithm. Formats for RSA and ECDH/ECDSA public keys are specified below.

If the key is provided as a URI, then the "uri" field MUST be present, containing a URI where the key can be retrieved, in the JSON format described above. The method that the recipient of a JSMS object uses to retrieve the key will depend on the URI scheme. For HTTP URIs, the relying party MUST issue an HTTP request with the GET method and an Accept header including the MIME type for JSMS PublicKey object, "[[MIMETYPE-TBD]]". For MAILTO, SIP, and XMPP URIs, the recipient MAY use the WebFinger protocol [I-D.hammer-webfinger] to retrieve a public key for the user.

If the key is referenced by an opaque identifier or "fingerprint", then the "id" field MUST be present, and contain the Base64-encoded SHA-1 hash of the public key, represented as a DER-encoded subjectPublicKeyInfo data structure. (This fingerprint value is the same as the one commonly included in the subjectKeyIdentifier field in an X.509 certificate.)

The recipient of a JSMS object can determine which of the above cases

a given key falls into by seeking the three fields in sequence. If a "type" field is present, then the key is represented directly. If a "uri" field is present, then the key is represented directly, but must be retrieved from the URI. Finally, if the "id" field is the only one of the three present, then the key is represented by ID only, and must be retrieved from somewhere else (e.g., from a certificate in the JSMS object).

Example: {"id": "ilLbR8FCEw-aiFcAAfUvpp75wdY="}

Example: {"uri": "xmpp:juliet@example.com"}

4.6.2.1. RSA Public Key

An RSA public key comprises two additional parameters in addition to the algorithm identifier "rsa".

"n": REQUIRED ByteString. The modulus, represented as an integer in network byte order (big-endian)

"e": REQUIRED Integer. The public exponent, represented as an integer in network byte order (big-endian)

Example: {"type":"rsa", "n":98739...04251, "e": 3}

4.6.2.2. Elliptic-Curve Public Key

Public keys for several types of elliptic curve algorithms, including ECDSA and ECDH, have the same format, namely an point on a specified elliptic curve. In an elliptic curve PublicKey object, the curve parameters are specified in the algorithm identifier, and there are two additional fields that specify the point on the curve:

"x": REQUIRED ByteString. The x coordinate of the point

"y": REQUIRED Integer. The y coordinate of the point. MUST be equal to 0 or 1.

These coordinates correspond to the compressed form of an elliptic curve point, as specified in [[SEC01]]. In terms of the calculation specified in section 2.3.3 of [[SEC01]], the "x" coordinate is the byte string X and the "y" coordinate is the reduced y coordinate (or, equivalently, Y mod 2).

Example: {"type":"ecdh",
"x":"IIIs_x1m6Na6xKN37vOwvy7AvFeG9HhBN2EN3u5EZQ4", "y": 1}

4.6.3. WrappedKey

In JSMS objects that use symmetric keys (for MAC or encryption), it is necessary for the originator to convey the symmetric key used for in JSMS computations to the recipient. The WrappedKey object is a JSON object that allows these keys to be provided either using key transport or key agreement. The following fields may be present in a WrappedKey object:

"type": REQUIRED TOKEN The type of wrapping being done. This document defines the following values for this field:

- "encryption": Symmetric key transport. The "KEKIdentifier" field MUST be present. Any other non-required fields MUST be ignored.
- "transport": Asymmetric key transport. The "recipientKey" field MUST be present. Any other non-required fields MUST be ignored.
- "agreement": Key agreement. The "originatorKey" and "recipientKey" MUST be present, and the "userKeyMaterial" field MAY be present. Any other non-required fields MUST be ignored.

"algorithm": REQUIRED AlgorithmIdentifier The algorithm used to encrypt the symmetric key

"encryptedKey": REQUIRED BYTES The symmetric key, encrypted according to the algorithm indicated by the "algorithm" value

"KEKIdentifier": OPTIONAL BYTES An opaque identifier for the symmetric key encryption key

"originatorKey": OPTIONAL PublicKey The public key of the originator

"recipientKey": OPTIONAL PublicKey The public key of the recipient

"userKeyMaterial": OPTIONAL BYTES User key material

The techniques used for wrapping and unwrapping the encrypted key is determined by "type" and "algorithm" fields. In general, the options are the same as for CMS [RFC5280], without the option for password-based key wrapping.

"encryption": The key is encrypted under a pre-shared symmetric key encryption key identified by the "KEKIdentifier" field

"transport": The key is encrypted under the recipient's public key, identified in the "recipientKey" field.

"agreement": The key is encrypted under a shared secret derived using a key agreement algorithm combining the originator's private key and the recipient's public key, corresponding to the "originatorKey" and "recipientKey", respectively. The value provided in the "userKeyMaterial" field may be used to provide additional entropy.

[[More detail to be added.]]

5. Compact Format

The compact JSON format of a JSMS object is identical to the normal JSMS format, except that field names are replaced with shorter equivalent field names. Translations for the field names above are given in the table below. In a given JSMS object, field names **MUST** either all be in long form or all be in short form. An implementation **MUST** reject a JSMS object with mixed long and short names as improperly formatted.

Common		Signature	
-----	-----	-----	-----
version	v	signatureAlgorithm	sa
type	t	key	k
signed	s	signature	sg
authenticated	au	-----	-----
encrypted	en	AlgorithmIdentifier	
content	c	-----	-----
-----	-----	name	nm
SignedData		-----	-----
-----	-----	PublicKey	
digestAlgorithm	da	-----	-----
signatures	ss	type	t
certificates	ce	id	i
certificatesURI	cu	uri	u
-----	-----	-----	-----
AuthenticatedData		WrappedKey	
-----	-----	-----	-----
algorithm	a	type	t
mac	mac	encryption	ec
keys	ks	transport	tr
keyId	ki	agreement	ag
-----	-----	algorithm	a
EncryptedData		encryptedKey	ek
-----	-----	KEKIdentifier	i
algorithm	a	originatorKey	o
keys	ks	recipientKey	r
mac	mac	userKeyMaterial	uk
-----	-----	-----	-----

In applications where a JSMS object is required to be URL-safe, it is **RECOMMENDED** that it be rendered in the compact serialization, then Base64-encoded.

```
[[ If there is a desire to avoid double-base64url-encoding things,
then we could define a mechanism for moving some fields out of the
object. ]]
```

6. Examples

This section contains complete examples of all three JSMS types. All white space is for readability only, and must be removed before the examples can be considered valid JSMS objects.

6.1. Parameters

RSA key:

```
{
  "type": "rsa",
  "n": "AfWGiFrDktMCi4LkD_vcIsqc0m4JSS0rNDk_5Zdi8fwja_qH0M7d3
U4tPUw7L0gPliSMakdTKX0S7uTV_v9FeY8_WrxDgbphrH9Zaz0PvTL
OuiKfRkMWK5A6nzl_PdP7_ujDWkvHKhWcJtM7irdn9K059X21EDtuq
GJyq7_v_c_",
  "e": "AQAB",
  "d": "EMwfyOqzfJQgZyhl_W40k8SpNdfgDpmqjBiPYubhLqIk7LZns6XDO3
7ZuLiZxT_WP04uMZ7UmV5URwUJVlxEpmfozhtLooCTPlowtRQQjhTa
Pzlf5nRKOhsO8e3PZY7O44ut2prWNNxYxDk52rH9GTECqGAmDNb1f
he6zX4KJk="
}
```

Key Tag: HK1RA8AQwcI=

Symmetric key: rQS8Dx6WQ_xDWTER8mAHnw==

Content:

"Attack at dawn!"

6.2. SignedData

In this object the content is signed under the specified RSA key pair, using SHA256 as the digest.

```

{
  "version": 1,
  "type": "signed",
  "digestAlgorithm": "sha256",
  "content": "QXR0YWNrIGF0IGRh24h",
  "signatures": [{
    "signatureAlgorithm": "rsa",
    "key": {
      "type": "rsa",
      "n": "AfWGInFrdktMCi4LkD_vcIsqc0m4JSS0rNDk_5Zdi8
fwja_qH0M7d3U4tPUw7L0gPlISmakdTKX0S7uTV_v9
FeY8_WrxDgbphrH9Zaz0PvTLOuiKfRkMWK5A6nz1_P
dP7_ujDWkvHKhWcJtM7irdn9K059X21EDtuqGJyq7_
v_c_",
      "e": "AQAB",
    },
    "signature": "AJll1tVYsRtGeHaJenAU-U3x4LxXklNoGrFwyu
xJWnYIeLZL16Ib7ZPvD79peMiSQAHAAdLKcI8e-
CpU6HNQ-MxE-tEXvaXOxuNZfVG9LBP9hq_ZwX
SguffHHzS9lLtVB0OzrXeszXtqD5igmecolA0E
8eabzujA4bdN6Umyc7rA"
  }]
}

```

6.3. AuthenticatedData

In this object the content is authenticated with a MAC under a randomly-generated key (AuthenticatedData Key above), wrapped using the key encryption key above, identified by the above key tag.

```

{
  "version": 1,
  "type": "authenticated",
  "algorithm": "hs256",
  "content": "QXR0YWNrIGF0IGRh24h",
  "mac": "990xwhrsX-COXUN0uF09HUHLU2CjdneeMqTtM4sGVdY=",
  "keys": [{
    "type": "encryption",
    "algorithm": "aes",
    "encryptedKey": "Dbf2O_ZIX0_Zfj-0aU6zQjn3xixj6vm7LVX
XFDdX4xqie5bZUSlennstIPYOyzzNx9Udt-J
LZZh-zM8A_FbsZ8zAibdJ3EPyd",
    "KEKIdentifier": "HK1RA8AQwcI="
  }]
}

```

As another example, the following object is a detached MAC (over the same content string) in the compact encoding. Here we use the key as the MAC key directly (instead of as a key encryption key). The

object is shown both in raw JSON form and in the Base64 encoding.

```
{ "v": 1, "t": "au", "a": "hs256", "ki": "HK1RA8AQwcI=",
  "mac": "PMVmhmrgrbj-KNybfMqHu4ySJ0GnVrwellMKpiuuG1IQ=" }
```

```
eyJ2IjogMSwidCI6ICJhdSIsImEiOiAiAiaHMyNTYiLCJraSI6ICJISzFSQThBUXdj
ST0iLA0KICAibWFjIjogIlBNVmlobXJnYmotS055YmZNcUh1NH1TSjBHblZyd2Ux
MU1LcGlldUdsSVE9In0=
```

6.4. EncryptedData

In this object, the content is encrypted under the general AEAD algorithm using AES-128-CBC for encryption and HMAC-SHA1 for authentication. The keys are described above as "EncryptedData Key (E)" and "EncryptedData Key (A)", respectively. The temporary keys are wrapped using the PKCS#1 wrapping, under the RSA key pair above.

```
{
  "version": 1,
  "type": "encrypted",
  "algorithm": {
    "name": "aes128-ccm",
    "n": "LTR8s7KKbd1Q1Q==",
    "m": 8
  },
  "content": "0nkXCLOVxM2oNJOSDCwASLTODIMVZQE=",
  "keys": [{
    "type": "transport",
    "algorithm": "rsaes-oaep",
    "encryptedKey": "AbAxRnd_u7lICJlBskq3kgQVs54RLMgOjNmALXF
      JjKqsQ4kLNL60VAoEswGOd2arGfcxoMCw9wMeSP
      FOIvOXGvSt2wJXR_6kwzOJv_YyTC_eZUJHpcLnr
      jKxB7Zf2_ap24W6JqcOYYVy2DhECcPgyvVRA_Ql
      ZNHFYdqaImgOKJv-",
    "recipientKey": {
      "type": "rsa",
      "n": "AfWGInFrdktMCi4LkD_vcIsqc0m4JSS0rNDk_5Zdi8fwja
        _qH0M7d3U4tPUw7L0gPliSMakdTKX0S7uTV_v9FeY8_Wrx
        DgbphrH9Zaz0PvTLOuiKfRkMWK5A6nz1_PdP7_ujDWkvHK
        hWcJtM7irdn9K059X21EDtuqGJyq7_v_c_",
      "e": "AQAB",
    }
  }]
}
```

7. Mapping to CMS

The JSMS message format is semantically equivalent to a profile of

the Cryptographic Message Syntax (CMS), and mirrors a fair bit of its syntactical structure as well. The top-level message types each map to top-level CMS types: SignedData to SignedData, AuthenticatedData to AuthenticatedData, and EncryptedData to AuthEnvelopedData [RFC5083]. The main difference other than encoding is that many optional fields have been removed, for example the protected and unprotected attributes.

This similarity also applies to the secondary objects. Just as in CMS, AlgorithmIdentifier objects carry an identifier for the algorithm (here a name instead of an OID) and any related parameters. The PublicKey object format is an amalgam of the SubjectKeyIdentifier from CMS and the SubjectPublicKeyInfo from X.509. PublicKey objects can be mapped to CMS constructs by converting them to SubjectKeyIdentifier objects (using the appropriate hash) and including a certificate containing the public key. The WrappedKey object format maps directly to the CMS RecipientInfo structure, with the above considerations related to public keys, and without the option for password-based wrapping.

The major way in which JSMS diverges from CMS is that it allows the use of static MAC keys, referenced by an identifier. CMS requires the use of random MAC keys, encrypted in a RecipientInfo (i.e., a WrappedKey) for each recipient. JSMS allows the use of random keys, but also includes the "keyId" field to reference static MAC keys directly. The security implications of this change are discussed in Section 10.

In fact, it should be possible to translate JSMS objects back and forth to CMS without changing any values (simply reformatting), with only a couple of exception cases:

- o JSMS objects that use static MAC keys cannot be translated to CMS because CMS does not allow this keying mechanism.
- o JSMS objects using general AEAD algorithms (according to [[draft-mcgrew-aead-aes-cbc-hmac-sha]]) because the required algorithm identifiers have not been defined for CMS.
- o CMS objects using features that are not supported in JSMS (e.g., password-based key wrapping) cannot be translated to JSMS.

8. Comparison to JWS/JWE/JWK

The overall JSMS structure covers the integrity, authentication, and encryption use cases as the JSON Web Encryption (JWE) and JSON Web Signature (JWS) specifications. Most of the fields in JWS and JWE map conceptually to JSMS fields, with a couple of exceptions. The major differences are as follows:

- o The signature and MAC functions of the JWS object are separated into SignedData and AuthenticatedData JSMS objects.
- o JSMS is pure JSON, whereas in JWE and JWS only the header parameters are represented in JSON.
- o JSMS parameters are not integrity-protected, as they are in JWE and JWS.
- o JSMS allows for full algorithm agility in key agreement, while JWE only allows ECDH-ES.
- o JSMS supports multiple recipients for EncryptedData and AuthenticatedData objects via the inclusion of multiple WrappedKey objects. Sending a JWE to multiple recipients requires re-encryption of the entire object for each recipient.
- o The "typ" and "zip" parameters are not defined in JSMS, but could be added without significant change.
- o JSMS requires that recipients MUST ignore unknown header parameters, in order to facilitate extensibility.

The PublicKey structure is analogous to the JSON Web Key (JWK) (with the public key parameters specified in the JSON Web Algorithms (JWA) document). The JWK "use" and "kid" parameters are not defined in JSMS, but could be added without significant change.

9. IANA Considerations

TODO:

- o Register MIME types
- o Registries for algorithms (signing, hash, MAC, encryption, encipherment, agreement)

10. Security Considerations

Much more to follow here.

```
[[ Given the CMS mapping above, import CMS security considerations.
]]
```

```
[[ Notes on identity for SignedData and AuthenticatedData: It is
important to note that the above verification process only checks
that the JSMS object was signed with a given public key. In order
for this information to be useful to an applications, it is usually
necessary to bind the public key to an application-layer identifier.
If the "certificates" or "certificatesURI" value is present, then the
recipient SHOULD verify that the chain is valid, and that the the
end-entity certificate chains to a trust anchor. In this case, the
recipient can consider the identity asserted in the end-entity
certificate to be bound to the public key. Applications using this
```

specification without certificates will need to specify an alternative mechanism for binding public keys to identifiers.]]

[[Notes on the security of static-key MACs. Need to periodically refresh keys.]]

[[For multiple signatures, the considerations of RFC 4853.]]

11. Acknowledgements

The inspirataion and starting point for this document was draft-rescorla-jsms-00. Thanks to Eric Rescorla and Joe Hildebrand for allowing me to re-use a fair bit of their document, and for some helpful early reviews.

12. References

12.1. Normative References

- [RFC2045] Freed, N. and N. Borenstein, "Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies", RFC 2045, November 1996.
- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.
- [RFC3339] Klyne, G., Ed. and C. Newman, "Date and Time on the Internet: Timestamps", RFC 3339, July 2002.
- [RFC3447] Jonsson, J. and B. Kaliski, "Public-Key Cryptography Standards (PKCS) #1: RSA Cryptography Specifications Version 2.1", RFC 3447, February 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.
- [RFC4648] Josefsson, S., "The Base16, Base32, and Base64 Data Encodings", RFC 4648, October 2006.
- [RFC5116] McGrew, D., "An Interface and Algorithms for Authenticated Encryption", RFC 5116, January 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.
- [RFC5649] Housley, R. and M. Dworkin, "Advanced Encryption Standard (AES) Key Wrap with Padding Algorithm", RFC 5649, September 2009.
- [RFC5840] Grewal, K., Montenegro, G., and M. Bhatia, "Wrapped Encapsulating Security Payload (ESP) for Traffic Visibility", RFC 5840, April 2010.
- [FIPS-180-3]
National Institute of Standards and Technology (NIST),
"Secure Hash Standard (SHS)", FIPS PUB 180-3,
October 2008.

12.2. Informative References

- [RFC3986] Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax", STD 66, RFC 3986, January 2005.
- [I-D.hammer-webfinger]
Hammer-Lahav, E., Fitzpatrick, B., and B. Cook, "The WebFinger Protocol", draft-hammer-webfinger-00 (work in progress), October 2009.
- [RFC5652] Housley, R., "Cryptographic Message Syntax (CMS)", STD 70, RFC 5652, September 2009.
- [RFC5083] Housley, R., "Cryptographic Message Syntax (CMS) Authenticated-Enveloped-Data Content Type", RFC 5083, November 2007.
- [I-D.ietf-jose-json-web-signature]
Jones, M., Bradley, J., and N. Sakimura, "JSON Web Signature (JWS)", draft-ietf-jose-json-web-signature-02 (work in progress), May 2012.
- [krawczyk-ate]
Krawczyk, H., "The Order of Encryption and Authentication for Protecting Communications (or: How Secure Is SSL?)", Advances in cryptology--CRYPTO 2001 August 2001.

[GCM] National Institute of Standards and Technology (NIST),
"Recommendation for Block Cipher Modes of Operation:
Galois/Counter Mode (GCM) and GMAC", SP 800-38D,
November 2007.

Appendix A. Acknowledgments

[TODO]

Author's Address

Richard Barnes
BBN Technologies
1300 N. 17th St.
Arlington, VA 22209
USA

Email: rbarnes@bbn.com

JOSE
Internet-Draft
Intended status: Informational
Expires: August 29, 2013

R. Barnes
BBN Technologies
February 25, 2013

Use Cases and Requirements for JSON Object Signing and Encryption (JOSE)
draft-barnes-jose-use-cases-02.txt

Abstract

Many Internet applications have a need for object-based security mechanisms in addition to security mechanisms at the network layer or transport layer. In the past, the Cryptographic Message Syntax has provided a binary secure object format based on ASN.1. Over time, the use of binary object encodings such as ASN.1 has been overtaken by text-based encodings, for example JavaScript Object Notation. This document defines a set of use cases and requirements for a secure object format encoded using JavaScript Object Notation, drawn from a variety of application security mechanisms currently in development.

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 August 29, 2013.

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	3
2. Definitions	4
3. Basic Requirements	5
4. Use Cases	6
4.1. Security Tokens and Authorization	6
4.2. XMPP	8
4.3. ALTO	11
4.4. Emergency Alerting	11
4.5. Web Cryptography	13
5. Requirements	14
5.1. Functional Requirements	14
5.2. Security Requirements	15
5.3. Desiderata	15
6. Acknowledgements	16
7. IANA Considerations	16
8. Security Considerations	16
9. References	16
9.1. Normative References	16
9.2. Informative References	18
Author's Address	19

1. Introduction

Internet applications rest on the layered architecture of the Internet, and take advantage of security mechanisms at all layers. Many applications rely primarily on channel-based security technologies, which create a secure channel at the IP layer or transport layer over which application data can flow [RFC4301][RFC5246]. These mechanisms, however, cannot provide end-to-end security in some cases. For example, in protocols with application-layer intermediaries, channel-based security protocols would protect messages from attackers between intermediaries, but not from the intermediaries themselves. These cases require object-based security technologies, which embed application data within a secure object that can be safely handled by untrusted entities.

The most well-known example of such a protocol today is the use of Secure/Multipurpose Internet Mail Extensions (S/MIME) protections within the email system [RFC5751][RFC5322]. An email message typically passes through a series of intermediate Mail Transfer Agents (MTAs) en route to its destination. While these MTAs often apply channel-based security protections to their interactions (e.g., [RFC3207]), these protections do not prevent the MTAs from interfering with the message. In order to provide end-to-end security protections in the presence of untrusted MTAs, mail users can use S/MIME to embed message bodies in a secure object format that can provide confidentiality, integrity, and data origin authentication.

S/MIME is based on the Cryptographic Message Syntax for secure objects (CMS) [RFC5652]. CMS is defined using Abstract Syntax Notation 1 (ASN.1) and traditionally encoded using the ASN.1 Distinguished Encoding Rules (DER), which define a binary encoding of the protected message and associated parameters [ITU.X690.1994]. In recent years, usage of ASN.1 has decreased (along with other binary encodings for general objects), while more applications have come to rely on text-based formats such as the Extensible Markup Language (XML) or the JavaScript Object Notation (JSON) [W3C.REC-xml-1998][RFC4627].

Many current applications thus have much more robust support for processing objects in these text-based formats than ASN.1 objects; indeed, many lack the ability to process ASN.1 objects at all. To simplify the addition of object-based security features to these applications, the IETF JSON Object Signing and Encryption (JOSE) working group has been chartered to develop a secure object format based on JSON. While the basic requirements for this object format are straightforward -- namely, confidentiality and integrity mechanisms, encoded in JSON -- early discussions in the working group

indicated that many applications hoping to use the formats define in JOSE have additional requirements. This document summarizes the use cases for JOSE envisioned by those applications and the resulting requirements for security mechanisms and object encoding.

Some systems that use XML have specified the use of XML-based security mechanisms for object security, namely XML Digital Signatures and XML Encryption [W3C.CR-xmlsig-core2-20120124][W3C.CR-xmlenc-core1-20120313]. These mechanisms are defined for use with several security token systems (e.g., SAML, WS-Federation, and OpenID connect [OASIS.saml-core-2.0-os][WS-Federation][OpenID.Messages]) and the CAP emergency alerting format [CAP]. In practice, however, XML-based secure object formats introduce similar levels of complexity to ASN.1, so developers that lack the tools or motivation to handle ASN.1 aren't able to use XML security either. This situation motivates the creation of a JSON-based secure object format that is simple enough to implement and deploy that it can be easily adopted by developers with minimal effort and tools.

2. Definitions

This document makes extensive use of standard security terminology [RFC4949]. In addition, because the use cases for JOSE and CMS are similar, we will sometimes make analogies to some CMS concepts [RFC5652].

The JOSE working group charter calls for the group to define three basic JSON object formats:

1. Confidentiality-protected object format
2. Integrity-protected object format
3. A format for expressing public keys

In the below, we will refer to these as the "encrypted object format", the "signed object format", and the "key format", respectively. In general, where there is no need to distinguish between asymmetric and symmetric operations, we will use the terms "signing", "signature", etc. to denote both true digital signatures involving asymmetric cryptography as well as message authentication codes using symmetric keys (MACs).

In the lifespan of a secure object, there are two basic roles, an entity that creates the object (e.g., encrypting or signing a payload), and an entity that uses the object (decrypting, verifying).

We will refer to these roles as "sender" and "recipient", respectively. Note that while some requirements and use cases may refer to these as single entities, each object may have multiple entities in each role. For example, a message may be signed by multiple senders, or decrypted by multiple recipients.

3. Basic Requirements

Obviously, for the encrypted and signed object formats, the necessary protections will be created using appropriate cryptographic mechanisms: symmetric or asymmetric encryption for confidentiality and MACs or digital signatures for integrity protection. In both cases, it is necessary for the JOSE format to support both symmetric and asymmetric operations.

- o The JOSE encrypted object format must support object encryption in the case where the sender and receiver share a symmetric key.
- o The JOSE encrypted object format must support object encryption in the case where the sender has only a public key for the receiver.
- o The JOSE signed object format must integrity protection using Message Authentication Codes (MACs), for the case where the sender and receiver share only a symmetric key.
- o The JOSE signed object format must integrity protection using digital signatures, for the case where the receiver has only a public key for the sender.

In cases where two entities are going to be exchanging several JOSE objects, it might be helpful to pre-negotiate some parameters so that they do not have to be signaled in every JOSE object. However, in order to not interfere with endpoints that do not support pre-negotiation, it is necessary to signal when pre-negotiated parameters are in use.

- o The JOSE signed and encrypted object formats must include a field that indicates that pre-negotiated parameters are to be used to process the object. This field may also provide an indication of which parameters are to be used.

The purpose of the key format is to provide the recipient with sufficient information to use the encoded key to process cryptographic messages. Thus it is necessary to include additional parameters along with the bare key.

- o The JOSE key format must include all algorithm parameters necessary to use the encoded key, including an identifier for the algorithm with which the key is used as well as any additional parameters required by the algorithm (e.g., elliptic curve parameters).

4. Use Cases

Based on early discussions of JOSE, several working groups developing application-layer protocols have expressed a desire to use JOSE in their designs for end-to-end security features. In this section, we summarize the use cases proposed by these groups and discuss the requirements that they imply for the JOSE object formats.

4.1. Security Tokens and Authorization

Security tokens are a common use case for object-based security, for example, SAML assertions [OASIS.saml-core-2.0-os]. Security tokens are used to convey information about a subject entity ("claims" or "assertions") from an issuer to a recipient. The security features of a token format enable the recipient to verify that the claims came from the issuer and, if the object is confidentiality-protected, that they were not visible to other parties.

Security tokens are used in federation protocols such as SAML 2.0, WS-Federation, and OpenID Connect [OASIS.saml-core-2.0-os][WS-Federation][OpenID.Messages], as well as in resource authorization protocols such as OAuth 2.0 [RFC6749]. In some cases, security tokens are used for client authentication and for access control [I-D.ietf-oauth-jwt-bearer][I-D.ietf-oauth-saml2-bearer].

The OAuth protocol defines a mechanism for distributing and using authorization tokens using HTTP [RFC6749]. A Client that wishes to access a protected resource requests authorization from the Resource Owner. If the Resource Owner allows this access, he directs an Authorization Server to issue an access token to the Client. When the Client wishes to access the protected resource, he presents the token to the relevant Resource Server, which verifies the validity of the token before providing access to the protected resource.

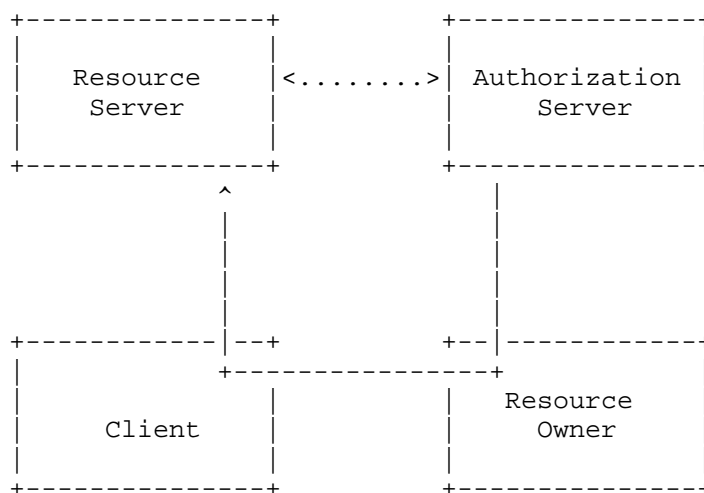


Figure 1: The OAuth process

In effect, this process moves the token from the Authorization Server (as a sender of the object) to the Resource Server (recipient), via the Client as well as the Resource Owner (the latter because of the HTTP mechanics underlying the protocol). So again we have a case where an application object is transported via untrusted intermediaries.

This application has two essential security requirements: Integrity and data origin authentication. Integrity protection is required so that the Resource Owner and the Client cannot modify the permission encoded in the token. Data origin authentication is required so that the Resource Server can verify that the token was issued by a trusted Authorization Server. Confidentiality protection may also be needed, if the Authorization Server is concerned about the visibility of permissions information to the Resource Owner or Client. For example, permissions related to social networking might be considered private information. Note, however, that OAuth already requires that the underlying HTTP transactions be protected by TLS, so confidentiality protection is not strictly necessary for this use case.

The confidentiality and integrity needs are met by the basic requirements for signed and encrypted object formats, whether the signing and encryption are provided using asymmetric or symmetric cryptography. The choice of which mechanism is applied will depend on the relationship between the two servers, namely whether they share a symmetric key or only public keys.

Authentication requirements will also depend on deployment characteristics. Where there is a relatively strong binding between the resource server and the authorization server, it may suffice for the Authorization Server issuing a token to be identified by the key used to sign the token. This requires that the token carry either the public key of the Authorization Server or an identifier for the public or symmetric key.

There may also be more advanced cases, where the Authorization Server's key is not known in advance to the Resource Server. This may happen, for instance, if an entity instantiated a collection of Authorization Servers (say for load balancing), each of which has an independent key pair. In these cases, it may be necessary to also include a certificate or certificate chain for the Authorization Server, so that the Resource Server can verify that the Authorization Server is an entity that it trusts.

The HTTP transport for OAuth imposes a particular constraint on the encoding. In the OAuth protocol, tokens frequently need to be passed as query parameters in HTTP URIs [RFC2616], after having been base64url encoded [RFC4648]. While there is no specified limit on the length of URIs (and thus of query parameters), in practice URIs of more than around 2,000 characters are rejected by some user agents. So this use case requires that a JOSE object have sufficiently small size even after signing, possibly encrypting, while still being simple to include in an HTTP URI query parameter.

Two related security token systems have similar requirements:

- o The JSON Web Token format (JWT) is a security token format based on JSON and JOSE [I-D.ietf-oauth-json-web-token]. It is used with both OpenID Connect and OAuth. Because JWTs are often used in contexts with limited space (e.g., HTTP query parameters), it is a core requirement for JWTs, and thus JOSE, to have a compact representation.
- o The OpenID Connect protocol is a simple, REST/JSON-based identity federation protocol layered on OAuth 2.0 [OpenID.Messages]. It uses the JWT and JOSE formats both to represent security tokens and to provide security for other protocol messages (signing and optionally encryption).

4.2. XMPP

The Extensible Messaging and Presence Protocol (XMPP) routes messages from one end client to another by way of XMPP servers [RFC6120]. There are typically two servers involved in delivering any given message: The first client (Alice) sends a message for another client

(B) to her server (A). Server A uses Bob's identity and the DNS to locate the server for Bob's domain (B), then delivers the message to that server. Server B then routes the message to Bob.

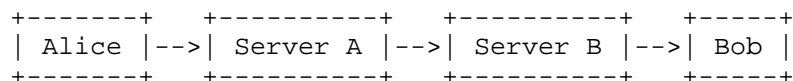


Figure 2: Delivering an XMPP message

The untrusted-intermediary problems are especially acute for XMPP because in many current deployments, the holder of an XMPP domain outsources the operation of the domain's servers to a different entity. In this environment, there is a clear risk of exposing the domain holder's private information to the domain operator. XMPP already has a defined mechanism for end-to-end security using S/MIME, but it has failed to gain widespread deployment [RFC3923], in part because of key management challenges and because of the difficulty of processing S/MIME objects.

The XMPP working group is in the process of developing a new end-to-end encryption system with an encoding based on JOSE and a clearer key management system [I-D.miller-xmpp-e2e]. The process of sending an encrypted message in this system involves two steps: First, the sender generates a symmetric Content Encryption Key (CEK), encrypts the message content, and sends the encrypted message to the desired set of recipients. Second, each recipient "dials back" to the sender, providing his public key; the sender then responds with the relevant CEK, wrapped with the recipient's public key.

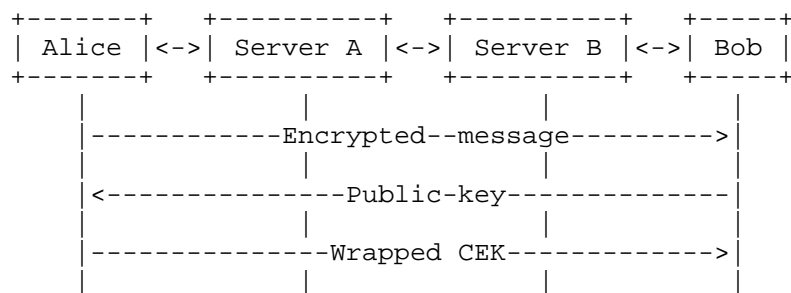


Figure 3: Delivering a secure XMPP message

The main thing that this system requires from the JOSE formats is confidentiality protection via content encryption, plus an integrity check via a MAC derived from the same symmetric key. The separation of the key exchange from the transmission of the encrypted content, however, requires that the JOSE encrypted object format allow wrapped

symmetric keys to be carried separately from the encrypted payload. In addition, the encrypted object will need to have a tag for the key that was used to encrypt the content, so that the recipient (Bob) can present the tag to the sender (Alice) when requesting the wrapped key.

Another important feature of XMPP is that it allows for the simultaneous delivery of a message to multiple recipients. In the diagrams above, Server A could deliver the message not only to Server B (for Bob) but also to Servers C, D, E, etc. for other users. In such cases, to avoid the multiple "dial back" transactions implied by the above mechanism, XMPP systems will likely cache public keys for end recipients, so that wrapped keys can be sent along with content on future messages. This implies that the JOSE encrypted object format must support the provision of multiple versions of the same wrapped CEK (much as a CMS EnvelopedData structure can include multiple RecipientInfo structures).

In the current draft of the XMPP end-to-end security system, each party is authenticated by virtue of the other party's trust in the XMPP message routing system. The sender is authenticated to the receiver because he can receive messages for the identifier "Alice" (in particular, the request for wrapped keys), and can originate messages for that identifier (the wrapped key). Likewise, the receiver is authenticated to the sender because he received the original encrypted message and originated the request for wrapped key. So the authentication here requires not only that XMPP routing be done properly, but also that TLS be used on every hop. Moreover, it requires that the TLS channels have strong authentication, since a man in the middle on any of the three hops can masquerade as Bob and obtain the key material for an encrypted message.

Because this authentication is quite weak (depending on the use of transport-layer security on three hops) and unverifiable by the endpoints, it is possible that the XMPP working group will integrate some sort of credentials for end recipients, in which case there would need to be a way to associate these credentials with JOSE objects.

Finally, it's worth noting that XMPP is based on XML, not JSON. So by using JOSE, XMPP will be carrying JSON objects within XML. It is thus a desirable property for JOSE objects to be encoded in such a way as to be safe for inclusion in XML. Otherwise, an explicit CDATA indication must be given to the parser to indicate that it is not to be parsed as XML. One way to meet this requirement would be to apply base64url encoding, but for XMPP messages of medium-to-large size, this could impose a fair degree of overhead.

4.3. ALTO

Application-Layer Traffic Optimization (ALTO) is a system for distributing network topology information to end devices, so that those devices can modify their behavior to have a lower impact on the network [I-D.ietf-alto-reqs]. The ALTO protocol distributes topology information in the form of JSON objects carried in HTTP [RFC2616][I-D.ietf-alto-protocol]. The basic version of ALTO is simply a client-server protocol, so simple use of HTTPS suffices for this case [RFC2818]. However, there is beginning to be some discussion of use cases for ALTO in which these JSON objects will be distributed through a collection of intermediate servers before reaching the client, while still preserving the ability of the client to authenticate the original source of the object. Even the base ALTO protocol notes that "ALTO clients obtaining ALTO information must be able to validate the received ALTO information to ensure that it was generated by an appropriate ALTO server."

In this case, the security requirements are straightforward. JOSE objects carrying ALTO payloads will need to bear digital signatures from the originating servers, which will be bound to certificates attesting to the identities of the servers. There is no requirement for confidentiality in this case, since ALTO information is generally public.

The more interesting questions are encoding questions. ALTO objects are likely to be much larger than payloads in the two cases above, with sizes of up to several megabytes. Processing of such large objects can be done more quickly if it can be done in a single pass, which may be possible if JOSE objects require specific orderings of fields within the JSON structure.

In addition, because ALTO objects are also encoded as JSON, they are already safe for inclusion in a JOSE object. Signed JOSE objects will likely carry the signed data in a string alongside the signature. JSON objects have the property that they can be safely encoded in JSON strings. All they require is that unnecessary white space be removed, a much simpler transformation than, say base64url encoding. This raises the question of whether it might be possible to optimize the JOSE encoding for certain "JSON-safe" cases.

4.4. Emergency Alerting

Emergency alerting is an emerging use case for IP networks [I-D.ietf-atoca-requirements]. Alerting systems allow authorities to warn users of impending danger by sending alert messages to connected devices. For example, in the event of hurricane or tornado, alerts might be sent to all devices in the path of the storm.

The most critical security requirement for alerting systems is that it must not be possible for an attacker to send false alerts to devices. Such a capability would potentially allow an attacker to create wide-spread panic. In practice, alert systems prevent these attacks both by controls on sending messages at points where alerts are originated, as well as by having recipients of alerts verify that the alert was sent by an authorized source. The former type of control implemented with local security on hosts from which alerts can be originated. The latter type implemented by digital signatures on alert messages (using channel-based or object-based mechanisms).

Alerts typically reach end recipients via a series of intermediaries. For example, while a national weather service might originate a hurricane alert, it might first be delivered to a national gateway, and then to network operators, who broadcast it to end subscribers.

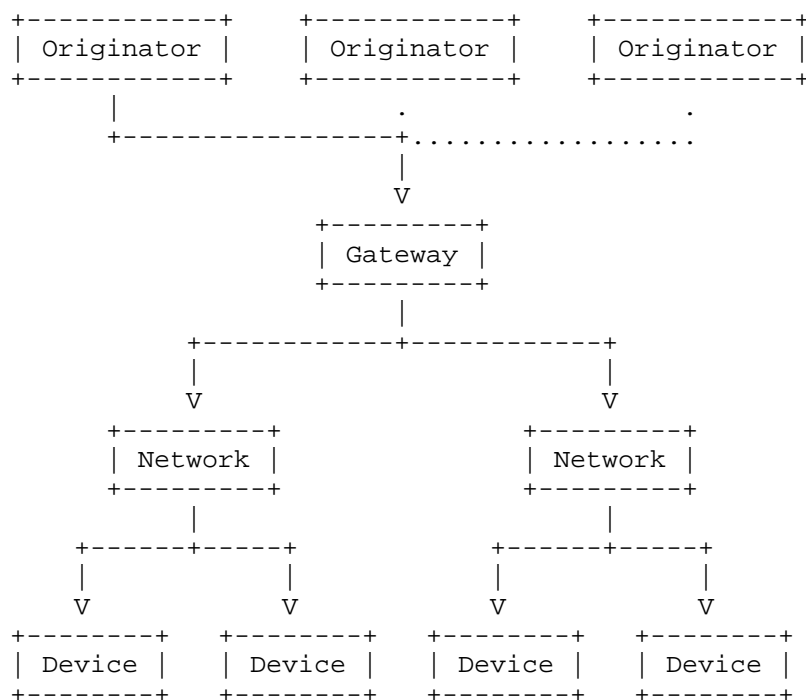


Figure 4: Delivering an emergency alert

In order to verify alert signatures, recipients must be provisioned with the proper public keys for trusted alert authorities. This trust may be "piece-wise" along the path the alert takes. For example, the alert relays operated by networks might have a full set

of certificates for all alert originators, while end devices may only trust their local alert relay. Or devices might require that a device be signed by an authorized originator and by its local network's relay.

This scenario creates a need for multiple signatures on alert documents, so that an alert can bear signatures from any or all of the entities that processed it along the path. In order to minimize complexity, these signatures should be "modular", in the sense that a new signature can be added without a need to alter or recompute previous signatures.

4.5. Web Cryptography

The W3C Web Cryptography API defines a standard cryptographic API for the web [WebCrypto]. If a browser exposes this API, then JavaScript provided as part of a web page can ask the browser to perform cryptographic operations, such as digest, MAC, encryption, or digital signing.

One of the key reasons to have the browser perform cryptographic operations is to avoid allowing JavaScript code to access the keying material used for these operations. For example, this separation would prevent code injected through a cross-site scripting (XSS) attack from reading and exfiltrating keys stored within a browser. While the malicious code could still use the key while running in the browser, this vulnerability can only be exercised while the vulnerable page is active in a user's browser.

However, the Web Cryptography API also provides a key export functionality, which can allow JavaScript to extract a key from the API in wrapped form. For example, the JavaScript might provide a public key for which the corresponding private key is held by another device. The wrapped key provided by the API could then be used to safely transport the key to the new device. While this could potentially allow malicious code to export a key, the need for an explicit export operation provides a control point, allowing for user notification or consent verification.

The Web Cryptography API also allows browsers to impose limitations on the usage of the keys it handles. For example, a symmetric key might be marked as usable only for encryption, and not for MAC. When a key is exported in wrapped form, these attributes should be carried along with it.

The Web Cryptography API thus requires formats to express several forms of keys. Obviously, the public key from an asymmetric key pair can be freely imported to and exported from the browser, so there

needs to be a format for public keys. There is also a need for a format to express private keys and symmetric keys. For non-public keys, the primary need is for a wrapped form, where the confidentiality and integrity of the key is assured cryptographically; these protections should also apply to any attributes of the key. It may also be useful to define a direct, unwrapped format, for use within a security boundary.

5. Requirements

This section summarizes the requirements from the above use cases, and lists further requirements not directly derived from the above use cases. There are also some constraints that are not hard requirements, but which are still desirable properties for the JOSE system to have.

5.1. Functional Requirements

F1 Define formats for secure objects that provide the following security properties:

- * Digital signature (integrity/authentication under an asymmetric key pair)
- * Message authentication (integrity/authentication under a symmetric key)
- * Encryption
- * Authenticated encryption

That is, the secure objects defined by this working group should provide equivalent security properties to the CMS SignedData, AuthenticatedData, EnvelopedData, and AuthEnveloped data objects [RFC5652] [RFC5083].

F2 Define a format for public keys and private keys for asymmetric cryptographic algorithms, with associated attributes, including a wrapped form for private keys.

F3 Define a format for symmetric keys with associated attributes, allowing for both wrapped and unwrapped keys.

F4 Define a JSON serialization for the above objects. An object in this encoding must be valid according to the JSON ABNF syntax [RFC4627]

- F5 Define a compact, URL-safe text serialization for the above objects.
- F6 Allow for attributes associated to wrapped keys to be bound to them cryptographically
- F7 Allow for wrapped keys to be separated from a secure object that uses a symmetric key. In such cases, cryptographic components of the secure object other than the wrapped key (e.g., ciphertext, MAC values) must be independent of the wrapped form of the key. For example, if an encrypted object is prepared for multiple recipients, then only the wrapped key may vary, not the ciphertext.

5.2. Security Requirements

- S1 Provide key management functions for all symmetric keys, including encryption keys and MAC keys. It should be possible to use any of the key management techniques provided in CMS [RFC5652]:
- * Key transport (wrapping for a public key)
 - * Key encipherment (wrapping for a symmetric key)
 - * Key agreement (wrapping for a DH public key)
 - * Password-based encryption (wrapping under a derived key)
- S2 Use cryptographic algorithms in a manner compatible with major validation processes. For example, if a FIPS standard allows algorithm A to be used for purpose X but not purpose Y, then JOSE should not recommend using algorithm A for purpose Y.
- S3 Support operation with or without pre-negotiation. It must be possible to create or process a secure object without any configuration beyond key provisioning. If it is possible to negotiate parameters out of band, then the object must signal that pre-negotiated parameters are to be used.

5.3. Desiderata

- D1 Maximize compatibility with the W3C WebCrypto specification, e.g., by using the same identifiers for algorithms.
- D2 Avoid JSON canonicalization to the extent possible. That is, all other things being equal, techniques that rely on fixing a serialization of an object (e.g., by base64url encoding it) are preferred over those that require converting an object to a

canonical form.

6. Acknowledgements

Thanks to Matt Miller for discussions related to XMPP end-to-end security model, and to Mike Jones for considerations related to security tokens and XML security. Thanks to Mark Watson for raising the need for representing symmetric keys and binding attributes to them.

7. IANA Considerations

This document makes no request of IANA.

8. Security Considerations

The primary focus of this document is the requirements for a JSON-based secure object format. At the level of general security considerations for object-based security technologies, the security considerations for this format are the same as for CMS [RFC5652]. The primary difference between the JOSE format and CMS is that JOSE is based on JSON, which does not have a canonical representation. The lack of a canonical form means that it is difficult to determine whether two JSON objects represent the same information, which could lead to vulnerabilities in some usages of JOSE.

9. References

9.1. Normative References

[I-D.ietf-alto-protocol]

Alimi, R., Penno, R., and Y. Yang, "ALTO Protocol", draft-ietf-alto-protocol-13 (work in progress), September 2012.

[I-D.ietf-alto-reqs]

Kiesel, S., Previdi, S., Stiernerling, M., Woundy, R., and Y. Yang, "Application-Layer Traffic Optimization (ALTO) Requirements", draft-ietf-alto-reqs-16 (work in progress), June 2012.

[I-D.ietf-atoca-requirements]

Schulzrinne, H., Norreys, S., Rosen, B., and H. Tschofenig, "Requirements, Terminology and Framework for

Exigent Communications", draft-ietf-atoca-requirements-03 (work in progress), March 2012.

[I-D.ietf-oauth-json-web-token]

Jones, M., Bradley, J., and N. Sakimura, "JSON Web Token (JWT)", draft-ietf-oauth-json-web-token-06 (work in progress), December 2012.

[I-D.miller-xmpp-e2e]

Miller, M., "End-to-End Object Encryption for the Extensible Messaging and Presence Protocol (XMPP)", draft-miller-xmpp-e2e-04 (work in progress), February 2013.

[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.

[RFC4949] Shirey, R., "Internet Security Glossary, Version 2", RFC 4949, August 2007.

[RFC5083] Housley, R., "Cryptographic Message Syntax (CMS) Authenticated-Enveloped-Data Content Type", RFC 5083, November 2007.

[RFC5652] Housley, R., "Cryptographic Message Syntax (CMS)", STD 70, RFC 5652, September 2009.

[RFC6120] Saint-Andre, P., "Extensible Messaging and Presence Protocol (XMPP): Core", RFC 6120, March 2011.

[RFC6749] Hardt, D., "The OAuth 2.0 Authorization Framework", RFC 6749, October 2012.

[W3C.CR-xmlsig-core2-20120124]

Datta, P., Hirsch, F., Eastlake, D., Cantor, S., Roessler, T., Reagle, J., Yiu, K., and D. Solo, "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-xml-1998]

Bray, T., Paoli, J., and C. Sperberg-McQueen, "Extensible Markup Language (XML) 1.0", W3C REC-xml-1998, February 1998,
<<http://www.w3.org/TR/1998/REC-xml-19980210/>>.

[WebCrypto]

Sleeve, R. and D. Dahl, "Web Cryptography API", January 2013.

9.2. Informative References

[CAP]

Botterell, A. and E. Jones, "Common Alerting Protocol v1.1", October 2005.

[I-D.ietf-oauth-jwt-bearer]

Jones, M., Campbell, B., and C. Mortimore, "JSON Web Token (JWT) Bearer Token Profiles for OAuth 2.0", draft-ietf-oauth-jwt-bearer-04 (work in progress), December 2012.

[I-D.ietf-oauth-saml2-bearer]

Campbell, B. and C. Mortimore, "SAML 2.0 Bearer Assertion Profiles for OAuth 2.0", draft-ietf-oauth-saml2-bearer-15 (work in progress), November 2012.

[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.

[OASIS.saml-core-2.0-os]

Cantor, S., Kemp, J., Philpott, R., and E. Maler, "Assertions and Protocol for the OASIS Security Assertion Markup Language (SAML) V2.0", OASIS Standard saml-core-2.0-os, March 2005.

[OpenID.Messages]

Sakimura, N., Bradley, J., Jones, M., de Medeiros, B., Mortimore, C., and E. Jay, "OpenID Connect Messages 1.0", June 2012,
<http://openid.net/specs/openid-connect-messages-1_0.html>.

- [RFC2616] Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., and T. Berners-Lee, "Hypertext Transfer Protocol -- HTTP/1.1", RFC 2616, June 1999.
- [RFC2818] Rescorla, E., "HTTP Over TLS", RFC 2818, May 2000.
- [RFC3207] Hoffman, P., "SMTP Service Extension for Secure SMTP over Transport Layer Security", RFC 3207, February 2002.
- [RFC3923] Saint-Andre, P., "End-to-End Signing and Object Encryption for the Extensible Messaging and Presence Protocol (XMPP)", RFC 3923, October 2004.
- [RFC4301] Kent, S. and K. Seo, "Security Architecture for the Internet Protocol", RFC 4301, December 2005.
- [RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, August 2008.
- [RFC5322] Resnick, P., Ed., "Internet Message Format", RFC 5322, October 2008.
- [RFC5751] Ramsdell, B. and S. Turner, "Secure/Multipurpose Internet Mail Extensions (S/MIME) Version 3.2 Message Specification", RFC 5751, January 2010.
- [WS-Federation] Kaler, C., McIntosh, M., Goodner, M., and A. Nadalin, "OpenID Connect Messages 1.0", May 2009, <<http://docs.oasis-open.org/wsfed/federation/v1.2/os/ws-federation-1.2-spec-os.html>>.

Author's Address

Richard Barnes
BBN Technologies
1300 N 17th St
Arlington, VA 22209
US

Email: rlb@ipv.sx

JOSE Working Group
Internet-Draft
Intended status: Standards Track
Expires: July 17, 2015

M. Jones
Microsoft
January 13, 2015

JSON Web Algorithms (JWA)
draft-ietf-jose-json-web-algorithms-40

Abstract

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

Status of this Memo

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

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

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

This Internet-Draft will expire on July 17, 2015.

Copyright Notice

Copyright (c) 2015 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	11
3.6. Using the Algorithm "none"	12
4. Cryptographic Algorithms for Key Management	12
4.1. "alg" (Algorithm) Header Parameter Values for JWE	12
4.2. Key Encryption with RSAES-PKCS1-V1_5	14
4.3. Key Encryption with RSAES OAEP	14
4.4. Key Wrapping with AES Key Wrap	15
4.5. Direct Encryption with a Shared Symmetric Key	16
4.6. Key Agreement with Elliptic Curve Diffie-Hellman Ephemeral Static (ECDH-ES)	16
4.6.1. Header Parameters Used for ECDH Key Agreement	17
4.6.1.1. "epk" (Ephemeral Public Key) Header Parameter . .	17
4.6.1.2. "apu" (Agreement PartyUInfo) Header Parameter . .	17
4.6.1.3. "apv" (Agreement PartyVInfo) Header Parameter . .	17
4.6.2. Key Derivation for ECDH Key Agreement	18
4.7. Key Encryption with AES GCM	19
4.7.1. Header Parameters Used for AES GCM Key Encryption . .	20
4.7.1.1. "iv" (Initialization Vector) Header Parameter . .	20
4.7.1.2. "tag" (Authentication Tag) Header Parameter . . .	20
4.8. Key Encryption with PBES2	20
4.8.1. Header Parameters Used for PBES2 Key Encryption . . .	21
4.8.1.1. "p2s" (PBES2 salt input) Parameter	21
4.8.1.2. "p2c" (PBES2 count) Parameter	21
5. Cryptographic Algorithms for Content Encryption	22
5.1. "enc" (Encryption Algorithm) Header Parameter Values for JWE	22
5.2. AES_CBC_HMAC_SHA2 Algorithms	23
5.2.1. Conventions Used in Defining AES_CBC_HMAC_SHA2	23
5.2.2. Generic AES_CBC_HMAC_SHA2 Algorithm	23
5.2.2.1. AES_CBC_HMAC_SHA2 Encryption	23
5.2.2.2. AES_CBC_HMAC_SHA2 Decryption	25
5.2.3. AES_128_CBC_HMAC_SHA_256	25
5.2.4. AES_192_CBC_HMAC_SHA_384	26
5.2.5. AES_256_CBC_HMAC_SHA_512	26
5.2.6. Content Encryption with AES_CBC_HMAC_SHA2	27
5.3. Content Encryption with AES GCM	27
6. Cryptographic Algorithms for Keys	28
6.1. "kty" (Key Type) Parameter Values	28

6.2.	Parameters for Elliptic Curve Keys	28
6.2.1.	Parameters for Elliptic Curve Public Keys	28
6.2.1.1.	"crv" (Curve) Parameter	29
6.2.1.2.	"x" (X Coordinate) Parameter	29
6.2.1.3.	"y" (Y Coordinate) Parameter	29
6.2.2.	Parameters for Elliptic Curve Private Keys	30
6.2.2.1.	"d" (ECC Private Key) Parameter	30
6.3.	Parameters for RSA Keys	30
6.3.1.	Parameters for RSA Public Keys	30
6.3.1.1.	"n" (Modulus) Parameter	30
6.3.1.2.	"e" (Exponent) Parameter	30
6.3.2.	Parameters for RSA Private Keys	31
6.3.2.1.	"d" (Private Exponent) Parameter	31
6.3.2.2.	"p" (First Prime Factor) Parameter	31
6.3.2.3.	"q" (Second Prime Factor) Parameter	31
6.3.2.4.	"dp" (First Factor CRT Exponent) Parameter	31
6.3.2.5.	"dq" (Second Factor CRT Exponent) Parameter	31
6.3.2.6.	"qi" (First CRT Coefficient) Parameter	31
6.3.2.7.	"oth" (Other Primes Info) Parameter	32
6.4.	Parameters for Symmetric Keys	32
6.4.1.	"k" (Key Value) Parameter	32
7.	IANA Considerations	33
7.1.	JSON Web Signature and Encryption Algorithms Registry	34
7.1.1.	Registration Template	34
7.1.2.	Initial Registry Contents	36
7.2.	Header Parameter Names Registration	42
7.2.1.	Registry Contents	42
7.3.	JSON Web Encryption Compression Algorithms Registry	43
7.3.1.	Registration Template	43
7.3.2.	Initial Registry Contents	44
7.4.	JSON Web Key Types Registry	44
7.4.1.	Registration Template	45
7.4.2.	Initial Registry Contents	45
7.5.	JSON Web Key Parameters Registration	46
7.5.1.	Registry Contents	46
7.6.	JSON Web Key Elliptic Curve Registry	48
7.6.1.	Registration Template	48
7.6.2.	Initial Registry Contents	49
8.	Security Considerations	50
8.1.	Cryptographic Agility	50
8.2.	Key Lifetimes	50
8.3.	RSAES-PKCS1-v1_5 Security Considerations	50
8.4.	AES GCM Security Considerations	50
8.5.	Unsecured JWS Security Considerations	51
8.6.	Denial of Service Attacks	51
8.7.	Reusing Key Material when Encrypting Keys	52
8.8.	Password Considerations	52
8.9.	Key Entropy and Random Values	53

8.10. Differences between Digital Signatures and MACs	53
8.11. Using Matching Algorithm Strengths	53
8.12. Adaptive Chosen-Ciphertext Attacks	53
8.13. Timing Attacks	53
8.14. RSA Private Key Representations and Blinding	53
9. Internationalization Considerations	53
10. References	53
10.1. Normative References	53
10.2. Informative References	55
Appendix A. Algorithm Identifier Cross-Reference	57
A.1. Digital Signature/MAC Algorithm Identifier Cross-Reference	58
A.2. Key Management Algorithm Identifier Cross-Reference . . .	58
A.3. Content Encryption Algorithm Identifier Cross-Reference .	59
Appendix B. Test Cases for AES_CBC_HMAC_SHA2 Algorithms	60
B.1. Test Cases for AES_128_CBC_HMAC_SHA_256	61
B.2. Test Cases for AES_192_CBC_HMAC_SHA_384	62
B.3. Test Cases for AES_256_CBC_HMAC_SHA_512	63
Appendix C. Example ECDH-ES Key Agreement Computation	64
Appendix D. Acknowledgements	66
Appendix E. Document History	67
Author's Address	78

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) [RFC7159] 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", "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 of [JWS].

UTF8(String) denotes the octets of the UTF-8 [RFC3629] representation of String, where String is a sequence of zero or more Unicode [UNICODE] characters.

ASCII(String) denotes the octets of the ASCII [RFC20] representation of String, where String is a sequence of zero or more ASCII characters.

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)", "Base64url Encoding", "Header Parameter", "JOSE Header", "JWS Payload", "JWS Protected Header", "JWS Signature", "JWS Signing Input", and "Unsecured JWS".

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

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 defined by the Internet Security Glossary, Version 2 [RFC4949] are incorporated into this specification: "Ciphertext", "Digital Signature", "Message Authentication Code (MAC)", and "Plaintext".

This term is defined by this specification:

Base64urlUInt

The representation of a positive or zero integer value 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 needed to represent the value. Zero is represented as BASE64URL(single zero-valued octet), which is "AA".

3. Cryptographic Algorithms for Digital Signatures and MACs

JWS uses cryptographic algorithms to digitally sign or create a Message Authentication Code (MAC) of the contents of the JWS Protected 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 Param Value	Digital Signature or MAC Algorithm	Implementation Requirements
HS256	HMAC using SHA-256	Required
HS384	HMAC using SHA-384	Optional
HS512	HMAC using SHA-512	Optional
RS256	RSASSA-PKCS-v1_5 using SHA-256	Recommended
RS384	RSASSA-PKCS-v1_5 using SHA-384	Optional
RS512	RSASSA-PKCS-v1_5 using SHA-512	Optional
ES256	ECDSA using P-256 and SHA-256	Recommended+
ES384	ECDSA using P-384 and SHA-384	Optional
ES512	ECDSA using P-521 and SHA-512	Optional
PS256	RSASSA-PSS using SHA-256 and MGF1 with SHA-256	Optional
PS384	RSASSA-PSS using SHA-384 and MGF1 with SHA-384	Optional
PS512	RSASSA-PSS using SHA-512 and MGF1 with SHA-512	Optional
none	No digital signature or MAC performed	Optional

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

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

3.2. HMAC with SHA-2 Functions

Hash-based Message Authentication Codes (HMACs) enable one to use a secret plus a cryptographic hash function to generate a Message Authentication Code (MAC). This can be used to demonstrate that whoever generated the MAC was in possession of the MAC key. The algorithm for implementing and validating HMACs is provided in RFC 2104 [RFC2104].

A key of the same size as the hash output (for instance, 256 bits for "HS256") or larger MUST be used with this algorithm. (This requirement is based on Section 5.3.4 (Security Effect of the HMAC Key) of NIST SP 800-117 [NIST.800-107], which states that the effective security strength is the minimum of the security strength of the key and two times the size of the internal hash value.)

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

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

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

The HMAC SHA-256 MAC for a JWS is validated by computing an HMAC value per RFC 2104, using SHA-256 as the hash algorithm "H", using the received JWS Signing Input as the "text" value, and using the shared key. This computed HMAC value is then compared to the result of base64url decoding the received encoded JWS Signature value. The comparison of the computed HMAC value to the JWS Signature value MUST be done in a constant-time manner to thwart timing attacks. Alternatively, the computed HMAC value can be base64url encoded and compared to the received encoded JWS Signature value (also in a constant-time manner), as this comparison produces the same result as comparing the unencoded values. In either case, if the values match, the HMAC has been validated.

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

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

3.3. Digital Signature with RSASSA-PKCS1-V1_5

This section defines the use of the RSASSA-PKCS1-V1_5 digital signature algorithm as defined in Section 8.2 of RFC 3447 [RFC3447] (commonly known as PKCS #1), using SHA-2 [SHS] hash functions.

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

The RSASSA-PKCS1-V1_5 SHA-256 digital signature is generated as follows: Generate a digital signature of the JWS Signing Input using

RSASSA-PKCS1-V1_5-SIGN and the SHA-256 hash function with the desired private key. This is the JWS Signature value.

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

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

The RSASSA-PKCS1-V1_5 SHA-256 digital signature for a JWS is validated as follows: Submit the JWS Signing Input, the JWS Signature, and the public key corresponding to the private key used by the signer to the RSASSA-PKCS1-V1_5-VERIFY algorithm using SHA-256 as the hash function.

Signing and validation with the RSASSA-PKCS1-V1_5 SHA-384 and RSASSA-PKCS1-V1_5 SHA-512 algorithms is performed identically to the procedure for RSASSA-PKCS1-V1_5 SHA-256 -- just using the corresponding hash algorithms instead of SHA-256.

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

3.4. Digital Signature with ECDSA

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

This specification defines the use of ECDSA with the P-256 curve and the SHA-256 cryptographic hash function, ECDSA with the P-384 curve and the SHA-384 hash function, and ECDSA with the P-521 curve and the SHA-512 hash function. The P-256, P-384, and P-521 curves are defined in [DSS].

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

1. Generate a digital signature of the JWS Signing Input using ECDSA P-256 SHA-256 with the desired private key. The output will be the pair (R, S), where R and S are 256 bit unsigned integers.

2. Turn R and S into octet sequences in big endian order, with each array being 32 octets long. The octet sequence representations MUST NOT be shortened to omit any leading zero octets contained in the values.
3. Concatenate the two octet sequences in the order R and then S. (Note that many ECDSA implementations will directly produce this concatenation as their output.)
4. The resulting 64 octet sequence is the JWS Signature value.

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

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

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

1. The JWS Signature value MUST be a 64 octet sequence. If it is not a 64 octet sequence, the validation has failed.
2. Split the 64 octet sequence into two 32 octet sequences. The first octet sequence represents R and the second S. The values R and S are represented as octet sequences using the Integer-to-OctetString Conversion defined in Section 2.3.7 of SEC1 [SEC1] (in big endian octet order).
3. Submit the JWS Signing Input R, S and the public key (x, y) to the ECDSA P-256 SHA-256 validator.

Signing and validation with the ECDSA P-384 SHA-384 and ECDSA P-521 SHA-512 algorithms is performed identically to the procedure for ECDSA P-256 SHA-256 -- just using the corresponding hash algorithms with correspondingly larger result values. For ECDSA P-384 SHA-384, R and S will be 384 bits each, resulting in a 96 octet sequence. For ECDSA P-521 SHA-512, R and S will be 521 bits each, resulting in a 132 octet sequence. (Note that the Integer-to-OctetString Conversion defined in Section 2.3.7 of SEC1 [SEC1] used to represent R and S as octet sequences adds zero-valued high-order padding bits when needed to round the size up to a multiple of 8 bits; thus, each 521-bit

integer is represented using 528 bits in 66 octets.)

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

3.5. Digital Signature with RSASSA-PSS

This section defines the use of the RSASSA-PSS digital signature algorithm as defined in Section 8.1 of RFC 3447 [RFC3447] with the MGF1 mask generation function and SHA-2 hash functions, always using the same hash function for both the RSASSA-PSS hash function and the MGF1 hash function. The size of the salt value is the same size as the hash function output. All other algorithm parameters use the defaults specified in Section A.2.3 of RFC 3447.

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

The RSASSA-PSS SHA-256 digital signature is generated as follows: Generate a digital signature of the JWS Signing Input using RSASSA-PSS-SIGN, the SHA-256 hash function, and the MGF1 mask generation function with SHA-256 with the desired private key. This is the JWS signature value.

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

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

The RSASSA-PSS SHA-256 digital signature for a JWS is validated as follows: Submit the JWS Signing Input, the JWS Signature, and the public key corresponding to the private key used by the signer to the RSASSA-PSS-VERIFY algorithm using SHA-256 as the hash function and using MGF1 as the mask generation function with SHA-256.

Signing and validation with the RSASSA-PSS SHA-384 and RSASSA-PSS SHA-512 algorithms is performed identically to the procedure for RSASSA-PSS SHA-256 -- just using the alternative hash algorithm in both roles.

3.6. Using the Algorithm "none"

JWSs MAY also be created that do not provide integrity protection. Such a JWS is called an Unsecured JWS. An Unsecured JWS uses the "alg" value "none" and is formatted identically to other JWSs, but MUST use the empty octet sequence as its JWS Signature value. Recipients MUST verify that the JWS Signature value is the empty octet sequence.

Implementations that support Unsecured JWSs 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 Unsecured JWSs by default. In order to mitigate downgrade attacks, applications MUST NOT signal acceptance of Unsecured JWSs at a global level, and SHOULD signal acceptance on a per-object basis. See Section 8.5 for security considerations associated with using this algorithm.

4. Cryptographic Algorithms for Key Management

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

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

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

alg Param Value	Key Management Algorithm	More Header Params	Implementation Requirements
RSA1_5	RSAES-PKCS1-V1_5	(none)	Recommended-
RSA-OAEP	RSAES OAEP using default parameters	(none)	Recommended+
RSA-OAEP-256	RSAES OAEP using SHA-256 and MGF1 with SHA-256	(none)	Optional
A128KW	AES Key Wrap with default initial value using 128 bit key	(none)	Recommended
A192KW	AES Key Wrap with default initial value using 192 bit key	(none)	Optional
A256KW	AES Key Wrap with default initial value using 256 bit key	(none)	Recommended
dir	Direct use of a shared symmetric key as the CEK	(none)	Recommended
ECDH-ES	Elliptic Curve Diffie-Hellman Ephemeral Static key agreement using Concat KDF	"epk", "apu", "apv"	Recommended+
ECDH-ES+A128KW	ECDH-ES using Concat KDF and CEK wrapped with "A128KW"	"epk", "apu", "apv"	Recommended
ECDH-ES+A192KW	ECDH-ES using Concat KDF and CEK wrapped with "A192KW"	"epk", "apu", "apv"	Optional
ECDH-ES+A256KW	ECDH-ES using Concat KDF and CEK wrapped with "A256KW"	"epk", "apu", "apv"	Recommended
A128GCMKW	Key wrapping with AES GCM using 128 bit key	"iv", "tag"	Optional

A192GCMKW	Key wrapping with AES GCM using 192 bit key	"iv", "tag"	Optional
A256GCMKW	Key wrapping with AES GCM using 256 bit key	"iv", "tag"	Optional
PBES2-HS256+A128KW	PBES2 with HMAC SHA-256 and "A128KW" wrapping	"p2s", "p2c"	Optional
PBES2-HS384+A192KW	PBES2 with HMAC SHA-384 and "A192KW" wrapping	"p2s", "p2c"	Optional
PBES2-HS512+A256KW	PBES2 with HMAC SHA-512 and "A256KW" wrapping	"p2s", "p2c"	Optional

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

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

See Appendix A.2 for a table cross-referencing the JWE "alg" (algorithm) values defined in this specification with the equivalent identifiers used by other standards and software packages.

4.2. Key Encryption with RSAES-PKCS1-V1_5

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

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

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

4.3. Key Encryption with RSAES OAEP

This section defines the specifics of encrypting a JWE CEK with RSAES using Optimal Asymmetric Encryption Padding (OAEP) [RFC3447]. Two sets of parameters for using OAEP are defined, which use different hash functions. In the first case, the default parameters specified by RFC 3447 in Section A.2.1 are used. (Those default parameters are the SHA-1 hash function and the MGF1 with SHA-1 mask generation function.) In the second case, the SHA-256 hash function and the

MGF1 with SHA-256 mask generation function are used.

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

alg Param Value	Key Management Algorithm
RSA-OAEP	RSAES OAEP using default parameters
RSA-OAEP-256	RSAES OAEP using SHA-256 and MGF1 with SHA-256

A key of size 2048 bits or larger MUST be used with these algorithms. (This requirement is based on Table 4 (Security-strength time frames) of NIST SP 800-57 [NIST.800-57], which requires 112 bits of security for new uses, and Table 2 (Comparable strengths) of the same, which states that 2048 bit RSA keys provide 112 bits of security.)

An example using RSAES OAEP with the default parameters is shown in Appendix A.1 of [JWE].

4.4. Key Wrapping with AES Key Wrap

This section defines the specifics of encrypting a JWE CEK with the Advanced Encryption Standard (AES) Key Wrap Algorithm [RFC3394] using the default initial value specified in Section 2.2.3.1.

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

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

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

4.5. Direct Encryption with a Shared Symmetric Key

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

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

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

This section defines the specifics of key agreement with Elliptic Curve Diffie-Hellman Ephemeral Static [RFC6090], in combination with the Concat KDF, as defined in Section 5.8.1 of [NIST.800-56A]. The key agreement result can be used in one of two ways:

1. directly as the Content Encryption Key (CEK) for the "enc" algorithm, in the Direct Key Agreement mode, or
2. as a symmetric key used to wrap the CEK with the "A128KW", "A192KW", or "A256KW" algorithms, in the Key Agreement with Key Wrapping mode.

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

In Direct Key Agreement mode, the output of the Concat KDF MUST be a key of the same length as that used by the "enc" algorithm. In this case, the empty octet sequence is used as the JWE Encrypted Key value. The "alg" Header Parameter value "ECDH-ES" is used in the Direct Key Agreement mode.

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

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

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

4.6.1. Header Parameters Used for ECDH Key Agreement

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

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

The "epk" (ephemeral public key) value created by the originator for the use in key agreement algorithms. This key is represented as a JSON Web Key [JWK] public key value. It MUST contain only public key parameters and SHOULD contain only the minimum JWK parameters necessary to represent the key; other JWK parameters included can be checked for consistency and honored or can be ignored. This Header Parameter MUST be present and MUST be understood and processed by implementations when these algorithms are used.

4.6.1.2. "apu" (Agreement PartyUInfo) Header Parameter

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

4.6.1.3. "apv" (Agreement PartyVInfo) Header Parameter

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

4.6.2. Key Derivation for ECDH Key Agreement

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

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

Z

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

keydatalen

This is set to the number of bits in the desired output key. For "ECDH-ES", this is length of the key used by the "enc" algorithm. For "ECDH-ES+A128KW", "ECDH-ES+A192KW", and "ECDH-ES+A256KW", this is 128, 192, and 256, respectively.

AlgorithmID

The *AlgorithmID* value is of the form *Datalen* || *Data*, where *Data* is a variable-length string of zero or more octets, and *Datalen* is a fixed-length, big endian 32 bit counter that indicates the length (in octets) of *Data*. In the Direct Key Agreement case, *Data* is set to the octets of the ASCII representation of the "enc" Header Parameter value. In the Key Agreement with Key Wrapping case, *Data* is set to the octets of the ASCII representation of the "alg" Header Parameter value.

PartyUInfo

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

PartyVInfo

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

the empty octet sequence.

SuppPubInfo

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

SuppPrivInfo

This is set to the empty octet sequence.

Applications need to specify how the "apu" and "apv" parameters are used for that application. The "apu" and "apv" values MUST be distinct, when used. Applications wishing to conform to [NIST.800-56A] need to provide values that meet the requirements of that document, e.g., by using values that identify the producer and consumer. 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 RFC 2631) and the "apv" field SHOULD NOT be present.

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

4.7. Key Encryption with AES GCM

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

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

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

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

The JWE Encrypted Key value is the Ciphertext output.

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

The following "alg" (algorithm) Header Parameter values are used to indicate that the JWE Encrypted Key is the result of encrypting the

CEK using the corresponding algorithm and key size:

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

4.7.1. Header Parameters Used for AES GCM Key Encryption

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

4.7.1.1. "iv" (Initialization Vector) Header Parameter

The "iv" (initialization vector) Header Parameter value is the base64url encoded representation of the 96 bit Initialization Vector value used for the key encryption operation. This Header Parameter MUST be present and MUST be understood and processed by implementations when these algorithms are used.

4.7.1.2. "tag" (Authentication Tag) Header Parameter

The "tag" (authentication tag) Header Parameter value is the base64url encoded representation of the 128 bit Authentication Tag value resulting from the key encryption operation. This Header Parameter MUST be present and MUST be understood and processed by implementations when these algorithms are used.

4.8. Key Encryption with PBES2

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

These algorithms use HMAC SHA-2 algorithms as the Pseudo-Random Function (PRF) for the PBKDF2 key derivation and AES Key Wrap [RFC3394] for the encryption scheme. The PBES2 password input is an octet sequence; if the password to be used is represented as a text string rather than an octet sequence, the UTF-8 encoding of the text string MUST be used as the octet sequence. The salt parameter MUST be computed from the "p2s" (PBES2 salt input) Header Parameter value and the "alg" (algorithm) Header Parameter value as specified in the "p2s" definition below. The iteration count parameter MUST be provided as the "p2c" Header Parameter value. The algorithms

respectively use HMAC SHA-256, HMAC SHA-384, and HMAC SHA-512 as the PRF and use 128, 192, and 256 bit AES Key Wrap keys. Their derived-key lengths respectively are 16, 24, and 32 octets.

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

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

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

4.8.1. Header Parameters Used for PBES2 Key Encryption

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

4.8.1.1. "p2s" (PBES2 salt input) Parameter

The "p2s" (PBES2 salt input) Header Parameter encodes a Salt Input value, which is used as part of the PBKDF2 salt value. The "p2s" value is `BASE64URL(Salt Input)`. This Header Parameter **MUST** be present and **MUST** be understood and processed by implementations when these algorithms are used.

The salt expands the possible keys that can be derived from a given password. A Salt Input value containing 8 or more octets **MUST** be used. A new Salt Input value **MUST** be generated randomly for every encryption operation; see RFC 4086 [RFC4086] for considerations on generating random values. The salt value used is `(UTF8(Alg) || 0x00 || Salt Input)`, where Alg is the "alg" Header Parameter value.

4.8.1.2. "p2c" (PBES2 count) Parameter

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

Parameter MUST be present and MUST be understood and processed by implementations when these algorithms are used.

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

5. Cryptographic Algorithms for Content Encryption

JWE uses cryptographic algorithms to encrypt and integrity protect the Plaintext and to also integrity protect additional authenticated data.

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

The table below is the set of "enc" (encryption algorithm) Header Parameter values that are defined by this specification for use with JWE.

enc Param Value	Content Encryption Algorithm	Implementation Requirements
A128CBC-HS256	AES_128_CBC_HMAC_SHA_256 authenticated encryption algorithm, as defined in Section 5.2.3	Required
A192CBC-HS384	AES_192_CBC_HMAC_SHA_384 authenticated encryption algorithm, as defined in Section 5.2.4	Optional
A256CBC-HS512	AES_256_CBC_HMAC_SHA_512 authenticated encryption algorithm, as defined in Section 5.2.5	Required
A128GCM	AES GCM using 128 bit key	Recommended
A192GCM	AES GCM using 192 bit key	Optional
A256GCM	AES GCM using 256 bit key	Recommended

All also use a JWE Initialization Vector value and produce JWE Ciphertext and JWE Authentication Tag values.

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

5.2. AES_CBC_HMAC_SHA2 Algorithms

This section defines a family of authenticated encryption algorithms built using a composition of Advanced Encryption Standard (AES) [AES] in Cipher Block Chaining (CBC) mode [NIST.800-38A] with PKCS #7 padding [RFC5652], Section 6.3 operations and HMAC [RFC2104, SHS] operations. This algorithm family is called AES_CBC_HMAC_SHA2. It also defines three instances of this family, the first using 128 bit CBC keys and HMAC SHA-256, the second using 192 bit CBC keys and HMAC SHA-384, and the third using 256 bit CBC keys and HMAC SHA-512. Test cases for these algorithms can be found in Appendix B.

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

5.2.1. Conventions Used in Defining AES_CBC_HMAC_SHA2

We use the following notational conventions.

CBC-PKCS5-ENC(X, P) denotes the AES CBC encryption of P using PKCS #7 padding using the cipher with the key X.

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

5.2.2. Generic AES_CBC_HMAC_SHA2 Algorithm

This section defines AES_CBC_HMAC_SHA2 in a manner that is independent of the AES CBC key size or hash function to be used. Section 5.2.2.1 and Section 5.2.2.2 define the generic encryption and decryption algorithms. Sections 5.2.3 through 5.2.5 define instances of AES_CBC_HMAC_SHA2 that specify those details.

5.2.2.1. AES_CBC_HMAC_SHA2 Encryption

The authenticated encryption algorithm takes as input four octet strings: a secret key K, a plaintext P, additional authenticated 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 additional authenticated 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.

The number of octets in the input key K MUST be the sum of MAC_KEY_LEN and ENC_KEY_LEN. The values of these parameters are specified by the Authenticated Encryption algorithms in Sections 5.2.3 through 5.2.5. 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 #7 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 the additional authenticated data A expressed as a 64-bit unsigned big endian integer.
5. A message authentication tag T is computed by applying HMAC [RFC2104] to the following data, in order:

the additional authenticated 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, additional authenticated data, initialization vector, and ciphertext, respectively.

MAC_KEY = initial MAC_KEY_LEN octets of K,

ENC_KEY = final ENC_KEY_LEN octets of K,

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

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

T = initial T_LEN octets of M.

5.2.2.2. AES_CBC_HMAC_SHA2 Decryption

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

1. The secondary keys MAC_KEY and ENC_KEY are generated from the input key K as in Step 1 of Section 5.2.2.1.
2. The integrity and authenticity of A and E are checked by computing an HMAC with the inputs as in Step 5 of Section 5.2.2.1. The value T, from the previous step, is compared to the first MAC_KEY length bits of the HMAC output. If those values are identical, then A and E are considered valid, and processing is continued. Otherwise, all of the data used in the MAC validation are discarded, and the Authenticated Encryption decryption operation returns an indication that it failed, and the operation halts. (But see Section 11.5 of [JWE] for security considerations on thwarting timing attacks.)
3. The value E is decrypted and the PKCS #7 padding is checked and removed. The value IV is used as the initialization vector. The value ENC_KEY is used as the decryption key.
4. The plaintext value is returned.

5.2.3. AES_128_CBC_HMAC_SHA_256

This algorithm is a concrete instantiation of the generic AES_CBC_HMAC_SHA2 algorithm above. It uses the HMAC message authentication code [RFC2104] with the SHA-256 hash function [SHS] to provide message authentication, with the HMAC output truncated to 128

bits, corresponding to the HMAC-SHA-256-128 algorithm defined in [RFC4868]. For encryption, it uses AES in the Cipher Block Chaining (CBC) mode of operation as defined in Section 6.2 of [NIST.800-38A], with PKCS #7 padding and a 128 bit initialization vector (IV) value.

The AES_CBC_HMAC_SHA2 parameters specific to AES_128_CBC_HMAC_SHA_256 are:

The input key K is 32 octets long.

ENC_KEY_LEN is 16 octets.

MAC_KEY_LEN is 16 octets.

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

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

5.2.4. AES_192_CBC_HMAC_SHA_384

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

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

ENC_KEY_LEN is 24 octets instead of 16.

MAC_KEY_LEN is 24 octets instead of 16.

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

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

5.2.5. AES_256_CBC_HMAC_SHA_512

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

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

ENC_KEY_LEN is 32 octets instead of 16.

MAC_KEY_LEN is 32 octets instead of 16.

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

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

5.2.6. Content Encryption with AES_CBC_HMAC_SHA2

This section defines the specifics of performing authenticated encryption with the AES_CBC_HMAC_SHA2 algorithms.

The CEK is used as the secret key K.

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

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

5.3. Content Encryption with AES GCM

This section defines the specifics of performing authenticated encryption with Advanced Encryption Standard (AES) in Galois/Counter Mode (GCM) [AES, NIST.800-38D].

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

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

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

6. Cryptographic Algorithms for Keys

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

6.1. "kty" (Key Type) Parameter Values

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

kty Param Value	Key Type	Implementation Requirements
EC	Elliptic Curve [DSS]	Recommended+
RSA	RSA [RFC3447]	Required
oct	Octet sequence (used to represent symmetric keys)	Required

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

6.2. Parameters for Elliptic Curve Keys

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

6.2.1. Parameters for Elliptic Curve Public Keys

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

elliptic curve public keys:

- o "crv"
- o "x"

The following member **MUST** also be present for elliptic curve public keys for the three curves defined in the following section:

- o "y"

6.2.1.1. "crv" (Curve) Parameter

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

- 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 7.6. Additional "crv" values can be registered by other specifications. Specifications registering additional curves must define what parameters are used to represent keys for the curves registered. The "crv" value is a case-sensitive string.

SEC1 [SEC1] point compression is not supported for any of these three curves.

6.2.1.2. "x" (X Coordinate) Parameter

The "x" (x coordinate) member contains the x coordinate for the elliptic curve point. It is represented as the base64url encoding of the octet string representation of the coordinate, as defined in Section 2.3.5 of SEC1 [SEC1]. The length of this octet string **MUST** be the full size of a coordinate for the curve specified in the "crv" parameter. For example, if the value of "crv" is "P-521", the octet string must be 66 octets long.

6.2.1.3. "y" (Y Coordinate) Parameter

The "y" (y coordinate) member contains the y coordinate for the elliptic curve point. It is represented as the base64url encoding of the octet string representation of the coordinate, as defined in Section 2.3.5 of SEC1 [SEC1]. The length of this octet string **MUST** be the full size of a coordinate for the curve specified in the "crv" parameter. For example, if the value of "crv" is "P-521", the octet string must be 66 octets long.

6.2.2. Parameters for Elliptic Curve Private Keys

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

6.2.2.1. "d" (ECC Private Key) Parameter

The "d" (ECC private key) member contains the Elliptic Curve private key value. It is represented as the base64url encoding of the octet string representation of the private key value, as defined in Section 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).

6.3. Parameters for RSA Keys

JWKs can represent RSA [RFC3447] keys. In this case, the "kty" member value is "RSA". The semantics of the parameters defined below are the same as those defined in Sections 3.1 and 3.2 of RFC 3447.

6.3.1. Parameters for RSA Public Keys

The following members **MUST** be present for RSA public keys.

6.3.1.1. "n" (Modulus) Parameter

The "n" (modulus) member contains the modulus value for the RSA public key. It is represented as a Base64urlUInt encoded value.

Note that implementers have found that some cryptographic libraries prefix an extra zero-valued octet to the modulus representations they return, for instance, returning 257 octets for a 2048 bit key, rather than 256. Implementations using such libraries will need to take care to omit the extra octet from the base64url encoded representation.

6.3.1.2. "e" (Exponent) Parameter

The "e" (exponent) member contains the exponent value for the RSA public key. It is represented as a Base64urlUInt encoded value.

For instance, when representing the value 65537, the octet sequence to be base64url encoded **MUST** consist of the three octets [1, 0, 1]; the resulting representation for this value is "AQAB".

6.3.2. Parameters for RSA Private Keys

In addition to the members used to represent RSA public keys, the following members are used to represent RSA private keys. The parameter "d" is REQUIRED for RSA private keys. The others enable optimizations and SHOULD be included by producers of JWKS representing RSA private keys. If the producer includes any of the other private key parameters, then all of the others MUST be present, with the exception of "oth", which MUST only be present when more than two prime factors were used.

6.3.2.1. "d" (Private Exponent) Parameter

The "d" (private exponent) member contains the private exponent value for the RSA private key. It is represented as a Base64urlUInt encoded value.

6.3.2.2. "p" (First Prime Factor) Parameter

The "p" (first prime factor) member contains the first prime factor. It is represented as a Base64urlUInt encoded value.

6.3.2.3. "q" (Second Prime Factor) Parameter

The "q" (second prime factor) member contains the second prime factor. It is represented as a Base64urlUInt encoded value.

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

The "dp" (first factor CRT exponent) member contains the Chinese Remainder Theorem (CRT) exponent of the first factor. It is represented as a Base64urlUInt encoded value.

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

The "dq" (second factor CRT exponent) member contains the Chinese Remainder Theorem (CRT) exponent of the second factor. It is represented as a Base64urlUInt encoded value.

6.3.2.6. "qi" (First CRT Coefficient) Parameter

The "qi" (first CRT coefficient) member contains the Chinese Remainder Theorem (CRT) coefficient of the second factor. It is represented as a Base64urlUInt encoded value.

6.3.2.7. "oth" (Other Primes Info) Parameter

The "oth" (other primes info) member contains an array of information about any third and subsequent primes, should they exist. When only two primes have been used (the normal case), this parameter MUST be omitted. When three or more primes have been used, the number of array elements MUST be the number of primes used minus two. For more information on this case, see the description of the OtherPrimeInfo parameters in Section A.1.2 of RFC 3447 [RFC3447], upon which the following parameters are modelled. If the consumer of a JWK does not support private keys with more than two primes and it encounters a private key that includes the "oth" parameter, then it MUST NOT use the key. Each array element MUST be an object with the following members:

6.3.2.7.1. "r" (Prime Factor)

The "r" (prime factor) parameter within an "oth" array member represents the value of a subsequent prime factor. It is represented as a Base64urlUInt encoded value.

6.3.2.7.2. "d" (Factor CRT Exponent)

The "d" (Factor CRT Exponent) parameter within an "oth" array member represents the CRT exponent of the corresponding prime factor. It is represented as a Base64urlUInt encoded value.

6.3.2.7.3. "t" (Factor CRT Coefficient)

The "t" (factor CRT coefficient) parameter within an "oth" array member represents the CRT coefficient of the corresponding prime factor. It is represented as a Base64urlUInt encoded value.

6.4. Parameters for Symmetric Keys

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

6.4.1. "k" (Key Value) Parameter

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

7. IANA Considerations

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

Values are registered on a Specification Required [RFC5226] basis after a three-week review period on the jose-reg-review@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 jose-reg-review@ietf.org mailing list for review and comment, with an appropriate subject (e.g., "Request to register algorithm: example").

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@ietf.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 description is clear.

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).

[[Note to the RFC Editor and IANA: Pearl Liang of ICANN had requested that the draft supply the following proposed registry description information. It is to be used for all registries established by this specification.

- o Protocol Category: JSON Object Signing and Encryption (JOSE)
- o Registry Location: <http://www.iana.org/assignments/jose>
- o Webpage Title: (same as the protocol category)
- o Registry Name: (same as the section title, but excluding the word "Registry", for example "JSON Web Signature and Encryption Algorithms")

]]

7.1. JSON Web Signature and Encryption Algorithms Registry

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

It is suggested that when multiple variations of algorithms are being registered that use keys of different lengths and the key lengths for each need to be fixed (for instance, because they will be created by key derivation functions), that the length of the key be included in the algorithm name. This allows readers of the JSON text to more easily make security decisions.

The Designated Expert(s) should perform reasonable due diligence that algorithms being registered are either currently considered cryptographically credible or are being registered as Deprecated or Prohibited.

The implementation requirements of an algorithm may be changed over time 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 after review by the Designated Experts(s), with the new specification defining the revised implementation requirements level.

7.1.1. Registration Template

Algorithm Name:

The name requested (e.g., "HS256"). This name is a case-sensitive ASCII string. Names may not match other registered names in a case-insensitive manner unless the Designated Expert(s) state that there is a compelling reason to allow an exception in this particular case.

Algorithm Description:

Brief description of the Algorithm (e.g., "HMAC using SHA-256").

Algorithm Usage Location(s):

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

JOSE Implementation Requirements:

The algorithm implementation requirements for JWS and JWE, which must be one the words Required, Recommended, Optional, Deprecated, or Prohibited. Optionally, the word can be followed by a "+" or "-". The use of "+" indicates that the requirement strength is likely to be increased in a future version of the specification. The use of "-" indicates that the requirement strength is likely to be decreased in a future version of the specification. Any identifiers registered for non-authenticated encryption algorithms or other algorithms that are otherwise unsuitable for direct use as JWS or JWE algorithms must be registered as "Prohibited".

Change Controller:

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

Specification Document(s):

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

Algorithm Analysis Documents(s):

References to publication(s) in well-known cryptographic conferences, by national standards bodies, or by other authoritative sources analyzing the cryptographic soundness of the algorithm to be registered. The designated experts may require convincing evidence of the cryptographic soundness of a new

algorithm to be provided with the registration request unless the algorithm is being registered as Deprecated or Prohibited. Having gone through working group and IETF review, the initial registrations made by this document are exempt from the need to provide this information.

7.1.2. Initial Registry Contents

- o Algorithm Name: "HS256"
- o Algorithm Description: HMAC using SHA-256
- o Algorithm Usage Location(s): "alg"
- o JOSE Implementation Requirements: Required
- o Change Controller: IESG
- o Specification Document(s): Section 3.1 of [[this document]]
- o Algorithm Analysis Documents(s): n/a

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

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

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

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

- o Algorithm Analysis Documents(s): n/a
- o Algorithm Name: "RS512"
- o Algorithm Description: RSASSA-PKCS-v1_5 using SHA-512
- o Algorithm Usage Location(s): "alg"
- o JOSE Implementation Requirements: Optional
- o Change Controller: IESG
- o Specification Document(s): Section 3.1 of [[this document]]
- o Algorithm Analysis Documents(s): n/a
- o Algorithm Name: "ES256"
- o Algorithm Description: ECDSA using P-256 and SHA-256
- o Algorithm Usage Location(s): "alg"
- o JOSE Implementation Requirements: Recommended+
- o Change Controller: IESG
- o Specification Document(s): Section 3.1 of [[this document]]
- o Algorithm Analysis Documents(s): n/a
- o Algorithm Name: "ES384"
- o Algorithm Description: ECDSA using P-384 and SHA-384
- o Algorithm Usage Location(s): "alg"
- o JOSE Implementation Requirements: Optional
- o Change Controller: IESG
- o Specification Document(s): Section 3.1 of [[this document]]
- o Algorithm Analysis Documents(s): n/a
- o Algorithm Name: "ES512"
- o Algorithm Description: ECDSA using P-521 and SHA-512
- o Algorithm Usage Location(s): "alg"
- o JOSE Implementation Requirements: Optional
- o Change Controller: IESG
- o Specification Document(s): Section 3.1 of [[this document]]
- o Algorithm Analysis Documents(s): n/a
- o Algorithm Name: "PS256"
- o Algorithm Description: RSASSA-PSS using SHA-256 and MGF1 with SHA-256
- o Algorithm Usage Location(s): "alg"
- o JOSE Implementation Requirements: Optional
- o Change Controller: IESG
- o Specification Document(s): Section 3.1 of [[this document]]
- o Algorithm Analysis Documents(s): n/a
- o Algorithm Name: "PS384"
- o Algorithm Description: RSASSA-PSS using SHA-384 and MGF1 with SHA-384

- o Algorithm Usage Location(s): "alg"
- o JOSE Implementation Requirements: Optional
- o Change Controller: IESG
- o Specification Document(s): Section 3.1 of [[this document]]
- o Algorithm Analysis Documents(s): n/a

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

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

- o Algorithm Name: "RSA1_5"
- o Algorithm Description: RSAES-PKCS1-V1_5
- o Algorithm Usage Location(s): "alg"
- o JOSE Implementation Requirements: Recommended-
- o Change Controller: IESG
- o Specification Document(s): Section 4.1 of [[this document]]
- o Algorithm Analysis Documents(s): n/a

- o Algorithm Name: "RSA-OAEP"
- o Algorithm Description: RSAES OAEP using default parameters
- o Algorithm Usage Location(s): "alg"
- o JOSE Implementation Requirements: Recommended+
- o Change Controller: IESG
- o Specification Document(s): Section 4.1 of [[this document]]
- o Algorithm Analysis Documents(s): n/a

- o Algorithm Name: "RSA-OAEP-256"
- o Algorithm Description: RSAES OAEP using SHA-256 and MGF1 with SHA-256
- o Algorithm Usage Location(s): "alg"
- o JOSE Implementation Requirements: Optional
- o Change Controller: IESG
- o Specification Document(s): Section 4.1 of [[this document]]

- o Algorithm Analysis Documents(s): n/a
- o Algorithm Name: "A128KW"
- o Algorithm Description: AES Key Wrap using 128 bit key
- o Algorithm Usage Location(s): "alg"
- o JOSE Implementation Requirements: Recommended
- o Change Controller: IESG
- o Specification Document(s): Section 4.1 of [[this document]]
- o Algorithm Analysis Documents(s): n/a
- o Algorithm Name: "A192KW"
- o Algorithm Description: AES Key Wrap using 192 bit key
- o Algorithm Usage Location(s): "alg"
- o JOSE Implementation Requirements: Optional
- o Change Controller: IESG
- o Specification Document(s): Section 4.1 of [[this document]]
- o Algorithm Analysis Documents(s): n/a
- o Algorithm Name: "A256KW"
- o Algorithm Description: AES Key Wrap using 256 bit key
- o Algorithm Usage Location(s): "alg"
- o JOSE Implementation Requirements: Recommended
- o Change Controller: IESG
- o Specification Document(s): Section 4.1 of [[this document]]
- o Algorithm Analysis Documents(s): n/a
- o Algorithm Name: "dir"
- o Algorithm Description: Direct use of a shared symmetric key
- o Algorithm Usage Location(s): "alg"
- o JOSE Implementation Requirements: Recommended
- o Change Controller: IESG
- o Specification Document(s): Section 4.1 of [[this document]]
- o Algorithm Analysis Documents(s): n/a
- o Algorithm Name: "ECDH-ES"
- o Algorithm Description: ECDH-ES using Concat KDF
- o Algorithm Usage Location(s): "alg"
- o JOSE Implementation Requirements: Recommended+
- o Change Controller: IESG
- o Specification Document(s): Section 4.1 of [[this document]]
- o Algorithm Analysis Documents(s): n/a
- o Algorithm Name: "ECDH-ES+A128KW"
- o Algorithm Description: ECDH-ES using Concat KDF and "A128KW" wrapping
- o Algorithm Usage Location(s): "alg"

- o JOSE Implementation Requirements: Recommended
- o Change Controller: IESG
- o Specification Document(s): Section 4.1 of [[this document]]
- o Algorithm Analysis Documents(s): n/a

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

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

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

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

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

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

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

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

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

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

- o Algorithm Name: "A256CBC-HS512"

- o Algorithm Description: AES_256_CBC_HMAC_SHA_512 authenticated encryption algorithm
- o Algorithm Usage Location(s): "enc"
- o JOSE Implementation Requirements: Required
- o Change Controller: IESG
- o Specification Document(s): Section 5.1 of [[this document]]
- o Algorithm Analysis Documents(s): n/a

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

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

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

7.2. Header Parameter Names Registration

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

7.2.1. Registry Contents

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

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

- o Header Parameter Name: "apv"
- o Header Parameter Description: Agreement PartyVInfo
- o Header Parameter Usage Location(s): JWE
- o Change Controller: IESG
- o Specification Document(s): Section 4.6.1.3 of [[this document]]

- o Header Parameter Name: "iv"
- o Header Parameter Description: Initialization Vector
- 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: "tag"
- o Header Parameter Description: Authentication Tag
- 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: "p2s"
- o Header Parameter Description: PBES2 salt
- 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: "p2c"
- o Header Parameter Description: PBES2 count
- o Header Parameter Usage Location(s): JWE
- o Change Controller: IESG
- o Specification Document(s): Section 4.8.1.2 of [[this document]]

7.3. JSON Web Encryption Compression Algorithms Registry

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

7.3.1. Registration Template

Compression Algorithm Value:

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

Compression Algorithm Description:

Brief description of the compression algorithm (e.g., "DEFLATE").

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

7.4. JSON Web Key Types Registry

This specification establishes the IANA JSON Web Key Types registry for values of the JWK "kty" (key type) parameter. The registry records the "kty" value, implementation requirements, and a reference to the specification that defines it.

The implementation requirements of a key type may be changed over time 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 after review by the Designated Experts(s), with the new specification defining the revised implementation requirements level.

7.4.1. Registration Template

"kty" Parameter Value:

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

Key Type Description:

Brief description of the Key Type (e.g., "Elliptic Curve").

Change Controller:

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

JOSE Implementation Requirements:

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

Specification Document(s):

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

7.4.2. Initial Registry Contents

This specification registers the values defined in Section 6.1.

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

- o "kty" Parameter Value: "RSA"

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

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

7.5. JSON Web Key Parameters Registration

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

7.5.1. Registry Contents

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

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

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

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

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

- o Parameter Name: "e"
- o Parameter Description: Exponent
- o Used with "kty" Value(s): "RSA"
- o Parameter Information Class: Public
- o Change Controller: IESG
- o Specification Document(s): Section 6.3.1.2 of [[this document]]

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

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

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

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

- o Parameter Name: "dq"
- o Parameter Description: Second Factor CRT Exponent
- o Used with "kty" Value(s): "RSA"
- o Parameter Information Class: Private

- o Change Controller: IESG
- o Specification Document(s): Section 6.3.2.5 of [[this document]]
- o Parameter Name: "qi"
- o Parameter Description: First CRT Coefficient
- o Used with "kty" Value(s): "RSA"
- o Parameter Information Class: Private
- o Change Controller: IESG
- o Specification Document(s): Section 6.3.2.6 of [[this document]]
- o Parameter Name: "oth"
- o Parameter Description: Other Primes Info
- o Used with "kty" Value(s): "RSA"
- o Parameter Information Class: Private
- o Change Controller: IESG
- o Specification Document(s): Section 6.3.2.7 of [[this document]]
- o Parameter Name: "k"
- o Parameter Description: Key Value
- o Used with "kty" Value(s): "oct"
- o Parameter Information Class: Private
- o Change Controller: IESG
- o Specification Document(s): Section 6.4.1 of [[this document]]

7.6. JSON Web Key Elliptic Curve Registry

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

The implementation requirements of a curve may be changed over time 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 after review by the Designated Experts(s), with the new specification defining the revised implementation requirements level.

7.6.1. Registration Template

Curve Name:

The name requested (e.g., "P-256"). Because a core goal of this specification is for the resulting representations to be compact, it is RECOMMENDED that the name be short -- not to exceed 8 characters without a compelling reason to do so. This name is case-sensitive. Names may not match other registered names in a

case-insensitive manner unless the Designated Expert(s) state that there is a compelling reason to allow an exception in this particular case.

Curve Description:

Brief description of the curve (e.g., "P-256 curve").

JOSE Implementation Requirements:

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

Change Controller:

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

Specification Document(s):

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

7.6.2. Initial Registry Contents

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

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

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

8. Security Considerations

All of the security issues that are pertinent to any cryptographic application must be addressed by JWS/JWE/JWK agents. Among these issues are protecting the user's asymmetric private and symmetric secret keys and employing countermeasures to various attacks.

The security considerations in [AES], [DSS], [JWE], [JWK], [JWS], [NIST.800-38D], [NIST.800-56A], [NIST.800-107], [RFC2104], [RFC3394], [RFC3447], [RFC5116], [RFC6090], and [SHS] apply to this specification.

8.1. Cryptographic Agility

Implementers should be aware that cryptographic algorithms become weaker with time. As new cryptanalysis techniques are developed and computing performance improves, the work factor to break a particular cryptographic algorithm will be reduced. Therefore, implementers and deployments must be prepared for the set of algorithms that are supported and used to change over time. Thus, cryptographic algorithm implementations should be modular, allowing new algorithms to be readily inserted.

8.2. Key Lifetimes

Many algorithms have associated security considerations related to key lifetimes and/or the number of times that a key may be used. Those security considerations continue to apply when using those algorithms with JOSE data structures. See NIST SP 800-57 [NIST.800-57] for specific guidance on key lifetimes.

8.3. RSAES-PKCS1-v1_5 Security Considerations

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

Keys used with RSAES-PKCS1-v1_5 must follow the constraints in Section 7.2 of RFC 3447. Also, keys with a low public key exponent value, as described in Section 3 of Twenty years of attacks on the RSA cryptosystem [Boneh99], must not be used.

8.4. AES GCM Security Considerations

Keys used with AES GCM must follow the constraints in Section 8.3 of [NIST.800-38D], which states: "The total number of invocations of the

authenticated encryption function shall not exceed 2^{32} , including all IV lengths and all instances of the authenticated encryption function with the given key". In accordance with this rule, AES GCM MUST NOT be used with the same key value more than 2^{32} times.

An Initialization Vector value MUST NOT ever be used multiple times with the same AES GCM key. One way to prevent this is to store a counter with the key and increment it with every use. The counter can also be used to prevent exceeding the 2^{32} limit above.

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

8.5. Unsecured JWS Security Considerations

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

An example means of preventing accepting Unsecured JWSs by default is for the "verify" method of a hypothetical JWS software library to have a Boolean "acceptUnsecured" 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 Unsecured JWS values if "none" is not in that list.

The following example illustrates the reasons for not accepting Unsecured JWSs at a global level. Suppose an application accepts JWSs over two channels, (1) HTTP and (2) HTTPS with client authentication. It requires a JWS signature on objects received over HTTP, but accepts Unsecured JWSs over HTTPS. If the application were to globally indicate that "none" is acceptable, then an attacker could provide it with an Unsecured JWS 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 "acceptUnsecured" to "true" for the first hypothetical JWS software library above), but not for each object received over HTTP.

8.6. 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 supply content using keys that would result in excessive

cryptographic processing, for example, keys larger than those mandated in this specification. Implementations should set and enforce upper limits on the key sizes they accept. Section 5.6.1 (Comparable Algorithm Strengths) of NIST SP 800-57 [NIST.800-57] contains statements on largest approved key sizes that may be applicable.

8.7. Reusing Key Material when Encrypting Keys

It is NOT RECOMMENDED to reuse the same entire set of 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 at least one new piece of key material for each encryption operation (e.g., a new Content Encryption Key, a new Initialization Vector, and/or a new PBES2 Salt), based on the considerations noted in this document as well as from RFC 4086 [RFC4086].

8.8. Password Considerations

Passwords are vulnerable to a number of attacks. To help mitigate some of these limitations, this document applies principles from RFC 2898 [RFC2898] to derive cryptographic keys from user-supplied passwords.

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

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

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

8.9. Key Entropy and Random Values

See Section 10.1 of [JWS] for security considerations on key entropy and random values.

8.10. Differences between Digital Signatures and MACs

See Section 10.5 of [JWS] for security considerations on differences between digital signatures and MACs.

8.11. Using Matching Algorithm Strengths

See Section 11.3 of [JWE] for security considerations on using matching algorithm strengths.

8.12. Adaptive Chosen-Ciphertext Attacks

See Section 11.4 of [JWE] for security considerations on adaptive chosen-ciphertext attacks.

8.13. Timing Attacks

See Section 10.9 of [JWS] and Section 11.5 of [JWE] for security considerations on timing attacks.

8.14. RSA Private Key Representations and Blinding

See Section 9.3 of [JWK] for security considerations on RSA private key representations and blinding.

9. Internationalization Considerations

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

10. References

10.1. Normative References

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

- [Boneh99] "Twenty years of attacks on the RSA cryptosystem", Notices of the American Mathematical Society (AMS), Vol. 46, No. 2, pp. 203-213 <http://crypto.stanford.edu/~dabo/pubs/papers/RSA-survey.pdf>, 1999.
- [DSS] National Institute of Standards and Technology, "Digital Signature Standard (DSS)", FIPS PUB 186-4, July 2013.
- [JWE] Jones, M. and J. Hildebrand, "JSON Web Encryption (JWE)", draft-ietf-jose-json-web-encryption (work in progress), January 2015.
- [JWK] Jones, M., "JSON Web Key (JWK)", draft-ietf-jose-json-web-key (work in progress), January 2015.
- [JWS] Jones, M., Bradley, J., and N. Sakimura, "JSON Web Signature (JWS)", draft-ietf-jose-json-web-signature (work in progress), January 2015.
- [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.
- [NIST.800-57] National Institute of Standards and Technology (NIST), "Recommendation for Key Management - Part 1: General (Revision 3)", NIST Special Publication 800-57, Part 1, Revision 3, July 2012.
- [RFC20] Cerf, V., "ASCII format for Network Interchange", RFC 20, October 1969.
- [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.
- [RFC3447] Jonsson, J. and B. Kaliski, "Public-Key Cryptography Standards (PKCS) #1: RSA Cryptography Specifications Version 2.1", RFC 3447, February 2003.
- [RFC3629] Yergeau, F., "UTF-8, a transformation format of ISO 10646", STD 63, RFC 3629, November 2003.
- [RFC4868] Kelly, S. and S. Frankel, "Using HMAC-SHA-256, HMAC-SHA-384, and HMAC-SHA-512 with IPsec", RFC 4868, May 2007.
- [RFC4949] Shirey, R., "Internet Security Glossary, Version 2", RFC 4949, August 2007.
- [RFC5652] Housley, R., "Cryptographic Message Syntax (CMS)", STD 70, RFC 5652, September 2009.
- [RFC6090] McGrew, D., Igoe, K., and M. Salter, "Fundamental Elliptic Curve Cryptography Algorithms", RFC 6090, February 2011.
- [RFC7159] Bray, T., "The JavaScript Object Notation (JSON) Data Interchange Format", RFC 7159, March 2014.
- [SEC1] Standards for Efficient Cryptography Group, "SEC 1: Elliptic Curve Cryptography", Version 2.0, May 2009.
- [SHS] National Institute of Standards and Technology, "Secure Hash Standard (SHS)", FIPS PUB 180-4, March 2012.
- [UNICODE] The Unicode Consortium, "The Unicode Standard", 1991-, <<http://www.unicode.org/versions/latest/>>.

10.2. Informative References

- [CanvasApp] Facebook, "Canvas Applications", 2010.
- [I-D.ietf-precis-saslprepbis]

Saint-Andre, P. and A. Melnikov, "Preparation, Enforcement, and Comparison of Internationalized Strings Representing Usernames and Passwords", draft-ietf-precis-saslprep-bis-13 (work in progress), December 2014.

[I-D.mcgregor-aead-aes-cbc-hmac-sha2]

McGrew, D., Foley, J., and K. Paterson, "Authenticated Encryption with AES-CBC and HMAC-SHA", draft-mcgregor-aead-aes-cbc-hmac-sha2-05 (work in progress), July 2014.

[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 (JCA) Reference Guide", 2014.

[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.800-107]

National Institute of Standards and Technology (NIST), "Recommendation for Applications Using Approved Hash Algorithms", NIST Special Publication 800-107, Revision 1, August 2012.

[NIST.800-63-1]

National Institute of Standards and Technology (NIST), "Electronic Authentication Guideline", NIST Special Publication 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.

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

[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.

[W3C.NOTE-xmldsig-core2-20130411]
Eastlake, D., Reagle, J., Solo, D., Hirsch, F., Roessler, T., Yiu, K., Datta, P., and S. Cantor, "XML Signature Syntax and Processing Version 2.0", World Wide Web Consortium Note NOTE-xmldsig-core2-20130411, April 2013, <<http://www.w3.org/TR/2013/NOTE-xmldsig-core2-20130411/>>.

[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>>.

[W3C.REC-xmlenc-core1-20130411]
Eastlake, D., Reagle, J., Hirsch, F., and T. Roessler, "XML Encryption Syntax and Processing Version 1.1", World Wide Web Consortium Recommendation REC-xmlenc-core1-20130411, April 2013, <<http://www.w3.org/TR/2013/REC-xmlenc-core1-20130411/>>.

Appendix A. Algorithm Identifier Cross-Reference

This appendix contains tables cross-referencing the cryptographic algorithm identifier values defined in this specification with the equivalent identifiers used by other standards and software packages. See XML DSIG [RFC3275], XML DSIG 2.0 [W3C.NOTE-xmldsig-core2-20130411], XML Encryption [W3C.REC-xmlenc-core-20021210], XML Encryption 1.1 [W3C.REC-xmlenc-core1-20130411], and Java Cryptography Architecture [JCA] for more information about the names defined by those documents.

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

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

JWS	XML DSIG	JCA	OID
HS256	http://www.w3.org/2001/04/xmldsig-more#hmac-sha256	HmacSHA256	1.2.840.1135.49.2.9
HS384	http://www.w3.org/2001/04/xmldsig-more#hmac-sha384	HmacSHA384	1.2.840.1135.49.2.10
HS512	http://www.w3.org/2001/04/xmldsig-more#hmac-sha512	HmacSHA512	1.2.840.1135.49.2.11
RS256	http://www.w3.org/2001/04/xmldsig-more#rsa-sha256	SHA256withRSA	1.2.840.1135.49.1.1.11
RS384	http://www.w3.org/2001/04/xmldsig-more#rsa-sha384	SHA384withRSA	1.2.840.1135.49.1.1.12
RS512	http://www.w3.org/2001/04/xmldsig-more#rsa-sha512	SHA512withRSA	1.2.840.1135.49.1.1.13
ES256	http://www.w3.org/2001/04/xmldsig-more#ecdsa-sha256	SHA256withECDSA	1.2.840.1004.5.4.3.2
ES384	http://www.w3.org/2001/04/xmldsig-more#ecdsa-sha384	SHA384withECDSA	1.2.840.1004.5.4.3.3
ES512	http://www.w3.org/2001/04/xmldsig-more#ecdsa-sha512	SHA512withECDSA	1.2.840.1004.5.4.3.4
PS256	http://www.w3.org/2007/05/xmldsig-more#sha256-rsa-MGF1	SHA256withRSAandMGF1	1.2.840.1135.49.1.1.10
PS384	http://www.w3.org/2007/05/xmldsig-more#sha384-rsa-MGF1	SHA384withRSAandMGF1	1.2.840.1135.49.1.1.10
PS512	http://www.w3.org/2007/05/xmldsig-more#sha512-rsa-MGF1	SHA512withRSAandMGF1	1.2.840.1135.49.1.1.10

A.2. Key Management Algorithm Identifier Cross-Reference

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

JWE	XML ENC	JCA	OID
RSA1_5	http://www.w3.org/2001/04/xmlenc#rsa-1_5	RSA/ECB/PKCS1Padding	1.2.840.113549.1.1.1
RSA-OAEP	http://www.w3.org/2001/04/xmlenc#rsa-oaep-mgf1p	RSA/ECB/OAEPWithSHA-1AndMGF1Padding	1.2.840.113549.1.1.7
RSA-OAEP-256	http://www.w3.org/2009/xmlenc11#rsa-oaep-mgf1sha256	RSA/ECB/OAEPWithSHA-256AndMGF1Padding & MGF1ParameterSpec.SHA256	1.2.840.113549.1.1.7
ECDH-ES	http://www.w3.org/2009/xmlenc11#ECDH-ES	ECDH	1.3.132.1.12
A128KW	http://www.w3.org/2001/04/xmlenc#kw-aes128	AESWrap	2.16.840.1.101.3.4.1.5
A192KW	http://www.w3.org/2001/04/xmlenc#kw-aes192	AESWrap	2.16.840.1.101.3.4.1.25
A256KW	http://www.w3.org/2001/04/xmlenc#kw-aes256	AESWrap	2.16.840.1.101.3.4.1.45

A.3. Content Encryption Algorithm Identifier Cross-Reference

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

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

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

A192GCM	http://www.w3.org/2009/ xmlenc11#aes192-gcm	AES/GCM/NoPa dding	2.16.840.1.10 1.3.4.1.26
A256GCM	http://www.w3.org/2009/ xmlenc11#aes256-gcm	AES/GCM/NoPa dding	2.16.840.1.10 1.3.4.1.46

Appendix B. Test Cases for AES_CBC_HMAC_SHA2 Algorithms

The following test cases can be used to validate implementations of the AES_CBC_HMAC_SHA2 algorithms defined in Section 5.2. They are also intended to correspond to test cases that may appear in a future version of [I-D.mcgrewe-aead-aes-cbc-hmac-sha2], demonstrating that the cryptographic computations performed are the same.

The variable names are those defined in Section 5.2. All values are hexadecimal.

B.1. Test Cases for AES_128_CBC_HMAC_SHA_256

AES_128_CBC_HMAC_SHA_256

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

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

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

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

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

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

AL =      00 00 00 00 00 00 01 50

E =      c8 0e df a3 2d df 39 d5 ef 00 c0 b4 68 83 42 79
        a2 e4 6a 1b 80 49 f7 92 f7 6b fe 54 b9 03 a9 c9
        a9 4a c9 b4 7a d2 65 5c 5f 10 f9 ae f7 14 27 e2
        fc 6f 9b 3f 39 9a 22 14 89 f1 63 62 c7 03 23 36
        09 d4 5a c6 98 64 e3 32 1c f8 29 35 ac 40 96 c8
        6e 13 33 14 c5 40 19 e8 ca 79 80 df a4 b9 cf 1b
        38 4c 48 6f 3a 54 c5 10 78 15 8e e5 d7 9d e5 9f
        bd 34 d8 48 b3 d6 95 50 a6 76 46 34 44 27 ad e5
        4b 88 51 ff b5 98 f7 f8 00 74 b9 47 3c 82 e2 db

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

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

B.2. Test Cases for AES_192_CBC_HMAC_SHA_384

```

K =      00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f
        10 11 12 13 14 15 16 17 18 19 1a 1b 1c 1d 1e 1f
        20 21 22 23 24 25 26 27 28 29 2a 2b 2c 2d 2e 2f

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

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

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

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

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

AL =     00 00 00 00 00 00 01 50

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

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

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

```

B.3. Test Cases for AES_256_CBC_HMAC_SHA_512

```

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

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

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

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

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

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

AL =     00 00 00 00 00 00 01 50

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

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

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

```

Appendix C. Example ECDH-ES Key Agreement Computation

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

In this example, a producer Alice is encrypting content to a consumer Bob. The producer (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 consumer'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": "VEmDZpDXXK8p8N0Cndsxs924q6nSlRXFASRl6BfUqdw"  
}
```

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 producer's (Alice's) ephemeral public key value to the consumer (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 following the octet sequence (using JSON array notation):
[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 ASCII string "A128GCM" - [65, 49, 50, 56, 71, 67, 77].

PartyUInfo

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

PartyVInfo

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

SuppPubInfo

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

SuppPrivInfo

This is set to the empty octet sequence.

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

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

Concatenating the round number 1 ([0, 0, 0, 1]), Z, and the OtherInfo value results in the Concat KDF round 1 hash input of:

[0, 0, 0, 1, 158, 86, 217, 29, 129, 113, 53, 211, 114, 131, 66, 131, 191, 132, 38, 156, 251, 49, 110, 163, 218, 128, 106, 72, 246, 218, 167, 121, 140, 254, 144, 196, 0, 0, 0, 7, 65, 49, 50, 56, 71, 67, 77, 0, 0, 0, 5, 65, 108, 105, 99, 101, 0, 0, 0, 3, 66, 111, 98, 0, 0, 0, 128]

The resulting derived key, which is the first 128 bits of the round 1 hash output is:

[86, 170, 141, 234, 248, 35, 109, 32, 92, 34, 40, 205, 113, 167, 16, 26]

The base64url encoded representation of this derived key is:

VqqN6vgjbsBcIijNcacQGg

Appendix D. Acknowledgements

Solutions for signing and encrypting JSON content were previously explored by Magic Signatures [MagicSignatures], JSON Simple Sign [JSS], Canvas Applications [CanvasApp], JSON Simple Encryption [JSE], and JavaScript Message Security Format [I-D.rescorla-jsms], all of which influenced this draft.

The Authenticated Encryption with AES-CBC and HMAC-SHA [I-D.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, Carsten Bormann, John Bradley, Brian Campbell, Alissa Cooper, Breno de Medeiros, Vladimir Dzhuvinov, Roni Even, Stephen Farrell, Yaron Y. Goland, Dick Hardt, Joe Hildebrand, Jeff Hodges, Edmund Jay, Charlie Kaufman, Barry Leiba, James Manger, Matt Miller, Kathleen Moriarty, Tony Nadalin, Axel Nennker, John Panzer, Emmanuel Raviart, Eric Rescorla, Pete Resnick, Nat Sakimura, Jim Schaad, Hannes Tschofenig, and Sean Turner.

Jim Schaad and Karen O'Donoghue chaired the JOSE working group and Sean Turner, Stephen Farrell, and Kathleen Moriarty 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]]

-40

- o Clarified the definitions of UTF8(String) and ASCII(String).

-39

- o Added the Algorithm Analysis Documents(s) field to the IANA JSON Web Signature and Encryption Algorithms registry.
- o Updated the reference to draft-ietf-precis-saslprepbis.

-38

- o Require discarding private keys with an "oth" parameter when the implementation does not support private keys with more than two primes.
- o Replaced uses of the phrases "JWS object" and "JWE object" with "JWS" and "JWE".

-37

- o Restricted algorithm names to using only ASCII characters.
- o Added language about ignoring private keys with an "oth" parameter when the implementation does not support private keys with more than two primes.
- o Updated the example IANA registration request subject line.

-36

- o Moved the normative "alg":"none" security considerations text into the algorithm definition.
- o Specified that registration reviews occur on the jose-reg-review@ietf.org mailing list.

-35

- o Addressed AppsDir reviews by Carsten Bormann.
- o Adjusted some table column widths.

-34

- o Addressed IESG review comments by Barry Leiba, Alissa Cooper, Pete Resnick, Stephen Farrell, and Richard Barnes.

-33

- o Changed the registration review period to three weeks.
- o Acknowledged additional contributors.

-32

- o Added a note to implementers about libraries that prefix an extra zero-valued octet to RSA modulus representations returned.
- o Addressed secdir review comments by Charlie Kaufman, Scott Kelly, and Stephen Kent.
- o Addressed Gen-ART review comments by Roni Even.
- o Replaced the term Plaintext JWS with Unsecured JWS.

-31

- o Referenced NIST SP 800-57 for guidance on key lifetimes.
- o Updated the reference to draft-mcgrew-aead-aes-cbc-hmac-sha2.

-30

- o Cleaned up the reference syntax in a few places.
- o Applied minor wording changes to the Security Considerations section.

-29

- o Replaced the terms JWS Header, JWE Header, and JWT Header with a single JOSE Header term defined in the JWS specification. This also enabled a single Header Parameter definition to be used and reduced other areas of duplication between specifications.

-28

- o Specified the use of PKCS #7 padding with AES CBC, rather than PKCS #5. (PKCS #7 is a superset of PKCS #5, and is appropriate for the 16 octet blocks used by AES CBC.)
- o Revised the introduction to the Security Considerations section. Also introduced additional subsection headings for security considerations items and moved a few security consideration items from here to the JWS and JWE drafts.

-27

- o Described additional security considerations.
- o Updated the JCA and XMLENC parameters for "RSA-OAEP-256" and the JCA parameters for "A128KW", "A192KW", "A256KW", and "ECDH-ES".

-26

- o Added algorithm identifier "RSA-OAEP-256" for RSAES OAEP using SHA-256 and MGF1 with SHA-256.
- o Clarified that the ECDSA signature values R and S are represented as octet sequences as defined in Section 2.3.7 of SEC1 [SEC1].
- o Noted that octet sequences are depicted using JSON array notation.
- o Updated references, including to W3C specifications.

-25

- o Corrected an external section number reference that had changed.

-24

- o Replaced uses of the term "associated data" wherever it was used to refer to a data value with "additional authenticated data", since both terms were being used as synonyms, causing confusion.
- o Updated the JSON reference to RFC 7159.

-23

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

-22

- o Corrected RFC 2119 terminology usage.
- o Replaced references to draft-ietf-json-rfc4627bis with RFC 7158.

-21

- o Compute the PBES2 salt parameter as (UTF8(Alg) || 0x00 || Salt Input), where the "p2s" Header Parameter encodes the Salt Input value and Alg is the "alg" Header Parameter value.
- o Changed some references from being normative to informative, addressing issue #90.

-20

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

-19

- o Used tables to show the correspondence between algorithm identifiers and algorithm descriptions and parameters in the algorithm definition sections, addressing issue #183.
- o Changed the "Implementation Requirements" registry field names to "JOSE Implementation Requirements" to make it clear that these implementation requirements apply only to JWS and JWE implementations.

-18

- o Changes to address editorial and minor issues #129, #134, #135, #158, #161, #185, #186, and #187.
- o Added and used Description registry fields.

-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.mcgregor-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: July 17, 2015

M. Jones
Microsoft
J. Hildebrand
Cisco
January 13, 2015

JSON Web Encryption (JWE)
draft-ietf-jose-json-web-encryption-40

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 July 17, 2015.

Copyright Notice

Copyright (c) 2015 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	6
3. JSON Web Encryption (JWE) Overview	8
3.1. JWE Compact Serialization Overview	9
3.2. JWE JSON Serialization Overview	9
3.3. Example JWE	10
4. JOSE Header	11
4.1. Registered Header Parameter Names	12
4.1.1. "alg" (Algorithm) Header Parameter	12
4.1.2. "enc" (Encryption Algorithm) Header Parameter	12
4.1.3. "zip" (Compression Algorithm) Header Parameter	13
4.1.4. "jku" (JWK Set URL) Header Parameter	13
4.1.5. "jwk" (JSON Web Key) Header Parameter	13
4.1.6. "kid" (Key ID) Header Parameter	13
4.1.7. "x5u" (X.509 URL) Header Parameter	13
4.1.8. "x5c" (X.509 Certificate Chain) Header Parameter	14
4.1.9. "x5t" (X.509 Certificate SHA-1 Thumbprint) Header Parameter	14
4.1.10. "x5t#S256" (X.509 Certificate SHA-256 Thumbprint) Header Parameter	14
4.1.11. "typ" (Type) Header Parameter	14
4.1.12. "cty" (Content Type) Header Parameter	14
4.1.13. "crit" (Critical) Header Parameter	14
4.2. Public Header Parameter Names	15
4.3. Private Header Parameter Names	15
5. Producing and Consuming JWEs	15
5.1. Message Encryption	15
5.2. Message Decryption	17
5.3. String Comparison Rules	20
6. Key Identification	20
7. Serializations	20
7.1. JWE Compact Serialization	20
7.2. JWE JSON Serialization	21
7.2.1. General JWE JSON Serialization Syntax	21
7.2.2. Flattened JWE JSON Serialization Syntax	24
8. TLS Requirements	24
9. Distinguishing between JWS and JWE Objects	25
10. IANA Considerations	25
10.1. JSON Web Signature and Encryption Header Parameters Registration	25

10.1.1. Registry Contents	25
11. Security Considerations	27
11.1. Key Entropy and Random Values	27
11.2. Key Protection	28
11.3. Using Matching Algorithm Strengths	28
11.4. Adaptive Chosen-Ciphertext Attacks	28
11.5. Timing Attacks	29
12. References	29
12.1. Normative References	29
12.2. Informative References	30
Appendix A. JWE Examples	31
A.1. Example JWE using RSAES OAEP and AES GCM	31
A.1.1. JOSE Header	31
A.1.2. Content Encryption Key (CEK)	31
A.1.3. Key Encryption	32
A.1.4. Initialization Vector	33
A.1.5. Additional Authenticated Data	33
A.1.6. Content Encryption	34
A.1.7. Complete Representation	34
A.1.8. Validation	35
A.2. Example JWE using RSAES-PKCS1-V1_5 and AES_128_CBC_HMAC_SHA_256	35
A.2.1. JOSE Header	35
A.2.2. Content Encryption Key (CEK)	36
A.2.3. Key Encryption	36
A.2.4. Initialization Vector	38
A.2.5. Additional Authenticated Data	38
A.2.6. Content Encryption	38
A.2.7. Complete Representation	39
A.2.8. Validation	39
A.3. Example JWE using AES Key Wrap and AES_128_CBC_HMAC_SHA_256	40
A.3.1. JOSE Header	40
A.3.2. Content Encryption Key (CEK)	40
A.3.3. Key Encryption	40
A.3.4. Initialization Vector	41
A.3.5. Additional Authenticated Data	41
A.3.6. Content Encryption	41
A.3.7. Complete Representation	42
A.3.8. Validation	42
A.4. Example JWE using General JWE JSON Serialization	43
A.4.1. JWE Per-Recipient Unprotected Headers	43
A.4.2. JWE Protected Header	43
A.4.3. JWE Unprotected Header	44
A.4.4. Complete JOSE Header Values	44
A.4.5. Additional Authenticated Data	44
A.4.6. Content Encryption	44
A.4.7. Complete JWE JSON Serialization Representation	45

A.5. Example JWE using Flattened JWE JSON Serialization	46
Appendix B. Example AES_128_CBC_HMAC_SHA_256 Computation	46
B.1. Extract MAC_KEY and ENC_KEY from Key	46
B.2. Encrypt Plaintext to Create Ciphertext	47
B.3. 64 Bit Big Endian Representation of AAD Length	47
B.4. Initialization Vector Value	48
B.5. Create Input to HMAC Computation	48
B.6. Compute HMAC Value	48
B.7. Truncate HMAC Value to Create Authentication Tag	48
Appendix C. Acknowledgements	48
Appendix D. Document History	49
Authors' Addresses	61

1. Introduction

JSON Web Encryption (JWE) represents encrypted content using JavaScript Object Notation (JSON) [RFC7159] based data structures. The JWE cryptographic mechanisms encrypt and provide integrity protection for an arbitrary sequence of octets.

Two closely related serializations for JWEs 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 JWEs 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", "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 of [JWS].

UTF8(String) denotes the octets of the UTF-8 [RFC3629] representation of String, where String is a sequence of zero or more Unicode [UNICODE] characters.

ASCII(String) denotes the octets of the ASCII [RFC20] representation of String, where String is a sequence of zero or more ASCII characters.

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)", "Base64url Encoding", "Collision-Resistant Name", "Header Parameter", "JOSE Header", and "StringOrURI".

These terms defined by the Internet Security Glossary, Version 2 [RFC4949] are incorporated into this specification: "Ciphertext", "Digital Signature", "Message Authentication Code (MAC)", and "Plaintext".

These terms are defined 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.

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 to produce the Ciphertext and the Authentication Tag.

JWE Encrypted Key

Encrypted Content Encryption Key (CEK) value. Note that for some algorithms, the JWE Encrypted Key value is specified as being the empty octet sequence.

JWE Initialization Vector

Initialization vector value 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 AAD

Additional value to be integrity protected by the authenticated encryption operation. This can only be present when using the JWE JSON Serialization. (Note that this can also be achieved when using either serialization by including the AAD value as an integrity protected Header Parameter value, but at the cost of the value being double base64url encoded.)

JWE Ciphertext

Ciphertext value resulting from authenticated encryption of the plaintext with additional authenticated data.

JWE Authentication Tag

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

JWE Protected Header

JSON object that contains the Header Parameters that are integrity protected by the authenticated encryption operation. These parameters apply to all recipients of the JWE. For the JWE Compact Serialization, this comprises the entire JOSE Header. For the JWE JSON Serialization, this is one component of the JOSE Header.

JWE Shared Unprotected Header

JSON object that contains the Header Parameters that apply to all recipients of the JWE that are not integrity protected. This can only be present when using the JWE JSON Serialization.

JWE Per-Recipient Unprotected Header

JSON object that contains Header Parameters that apply to a single recipient of the JWE. These Header Parameter values are not integrity protected. This can only be present when using the JWE JSON Serialization.

JWE Compact Serialization

A representation of the JWE as a compact, URL-safe string.

JWE JSON Serialization

A representation of the JWE as a JSON object. The JWE JSON Serialization enables the same content to be encrypted to multiple parties. This representation is neither optimized for compactness 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. These JSON data structures MAY contain white space and/or line breaks before or after any JSON values or structural characters, in accordance with Section 2 of RFC 7159 [RFC7159]. A JWE represents these logical values (each of which is defined in Section 2):

- o JOSE Header
- o JWE Encrypted Key
- o JWE Initialization Vector
- o JWE AAD

- o JWE Ciphertext
- o JWE Authentication Tag

For a JWE, the JOSE Header members are the union of the members of these values (each of which is defined in Section 2):

- o JWE Protected Header
- o JWE Shared Unprotected Header
- o JWE Per-Recipient Unprotected Header

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.

This document defines two serializations for JWEs: 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, since JSON lacks a way to directly represent arbitrary octet sequences. When present, the JWE AAD is also base64url encoded.

3.1. JWE Compact Serialization Overview

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

In the JWE Compact Serialization, a JWE is represented as the concatenation:

```
BASE64URL(UTF8(JWE Protected Header)) || '.' ||  
BASE64URL(JWE Encrypted Key) || '.' ||  
BASE64URL(JWE Initialization Vector) || '.' ||  
BASE64URL(JWE Ciphertext) || '.' ||  
BASE64URL(JWE Authentication Tag)
```

See Section 7.1 for more information about the JWE Compact Serialization.

3.2. JWE JSON Serialization Overview

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 JOSE Header are the union 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 is represented as a JSON object containing some or all of these eight members:

- "protected", with the value `BASE64URL(UTF8(JWE Protected Header))`
- "unprotected", with the value `JWE Shared Unprotected Header`
- "header", with the value `JWE Per-Recipient Unprotected Header`
- "encrypted_key", with the value `BASE64URL(JWE Encrypted Key)`
- "iv", with the value `BASE64URL(JWE Initialization Vector)`
- "ciphertext", with the value `BASE64URL(JWE Ciphertext)`
- "tag", with the value `BASE64URL(JWE Authentication Tag)`
- "aad", with the value `BASE64URL(JWE AAD)`

The six base64url encoded result strings and the two unprotected JSON object values are 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.

3.3. Example JWE

This example encrypts the plaintext "The true sign of intelligence is not knowledge but imagination." to the recipient.

The following example JWE Protected Header declares that:

- o The Content Encryption Key is encrypted to the recipient using the RSAES OAEP [RFC3447] algorithm to produce the JWE Encrypted Key.
- o Authenticated encryption is performed on the Plaintext using the AES GCM [AES, NIST.800-38D] algorithm with a 256 bit key to produce the Ciphertext and the Authentication Tag.

```
{"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 Perform authenticated encryption on the Plaintext with the AES GCM algorithm 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.
OKOawDo13gRp2ojaHV7LFpZcgV7T6DVZKTyKOMTYUmKoTCVJRgckCL9kiMT03JGe
ipsEdY3mx_etLbbWSrFr05kLzcSr4qKAq7YN7e9jwQRb23nfa6c9d-StnImGyFDb
Sv04uVuxIp5ZmslgNxKKK2Da14B8S4rzVRltdYwam_lDp5XnZAYpQdb76FdIKLaV
mqqfwX7XWRxv2322i-vDxRfqNzo_tETKzpVLzfiwQyeyPGLBIO56YJ7eObdv0je8
1860ppamavo35UgoRdbYaBcoh9QcfylQr66oc6vFWXRcZ_ZT2LawVCWTIy3brGPi
6UklfCpIMfIjf7iGdXKHZg.
48V1_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, including examples using the JWE JSON Serialization in Sections A.4 and A.5.

4. JOSE Header

For a JWE, the members of the JSON object(s) representing the JOSE Header describe the encryption applied to the Plaintext and optionally additional properties of the JWE. The Header Parameter names within the JOSE Header MUST be unique, just as described in Section 4 of [JWS]. The rules about handling Header Parameters that are not understood by the implementation are also the same. The classes of Header Parameter names are likewise the same.

4.1. Registered Header Parameter Names

The following Header Parameter names for use in JWEs 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 Algorithm) Header Parameter

The "enc" (encryption algorithm) Header Parameter identifies the content encryption algorithm used to perform authenticated encryption on the Plaintext to produce the Ciphertext and the Authentication Tag. This algorithm MUST be an AEAD algorithm with a specified key length. The encrypted content is not usable 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 ASCII string containing a StringOrURI value. This Header Parameter MUST be present and 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 5.1 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. When used, this Header Parameter MUST be integrity protected; therefore, it MUST occur only within the JWE Protected Header. 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. "kid" (Key ID) Header Parameter

This parameter has the same meaning, syntax, and processing rules as the "kid" Header Parameter defined in Section 4.1.4 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.

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

This parameter has the same meaning, syntax, and processing rules as the "x5u" Header Parameter defined in Section 4.1.5 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.

4.1.8. "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. "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 4.1.7 of [JWS], except that the 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. Note that certificate thumbprints are also sometimes known as certificate fingerprints.

4.1.10. "x5t#S256" (X.509 Certificate SHA-256 Thumbprint) Header Parameter

This parameter has the same meaning, syntax, and processing rules as the "x5t#S256" Header Parameter defined in Section 4.1.8 of [JWS], except that the 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. Note that certificate thumbprints are also sometimes known as certificate fingerprints.

4.1.11. "typ" (Type) Header Parameter

This parameter has the same meaning, syntax, and processing rules as the "typ" Header Parameter defined in Section 4.1.9 of [JWS], except that the type is that of this complete JWE.

4.1.12. "cty" (Content Type) Header Parameter

This parameter has the same meaning, syntax, and processing rules as the "cty" Header Parameter defined in Section 4.1.10 of [JWS], except that the type is that of the secured content (the plaintext).

4.1.13. "crit" (Critical) Header Parameter

This parameter has the same meaning, syntax, and processing rules as the "crit" Header Parameter defined in Section 4.1.11 of [JWS],

except that Header Parameters for a JWE are being referred to, rather than Header Parameters for a JWS.

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. 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 (steps 1-7) 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. If a "zip" parameter was included, compress the Plaintext using the specified compression algorithm and let M be the octet sequence representing the compressed Plaintext; otherwise, let M be the octet sequence representing the Plaintext.
12. Create the JSON object(s) containing the desired set of Header Parameters, which together comprise the JOSE Header: if the JWE Compact Serialization is being used, the JWE Protected Header, or if the JWE JSON Serialization is being used, one or more of the JWE Protected Header, the JWE Shared Unprotected Header, and the JWE Per-Recipient Unprotected Header.
13. Compute the Encoded Protected 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.
14. Let the Additional Authenticated Data encryption parameter be `ASCII(Encoded Protected Header)`. However if a JWE AAD value is

present (which can only be the case when using the JWE JSON Serialization), instead let the Additional Authenticated Data encryption parameter be ASCII(Encoded Protected Header || '.' || BASE64URL(JWE AAD)).

15. 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).
16. Compute the encoded ciphertext value BASE64URL(JWE Ciphertext).
17. Compute the encoded authentication tag value BASE64URL(JWE Authentication Tag).
18. If a JWE AAD value is present, compute the encoded AAD value BASE64URL(JWE AAD).
19. 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.

When there are multiple recipients, it is an application decision which of the 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 considered invalid. 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 considered invalid.

1. Parse the JWE representation to extract the serialized values for the components of the JWE. When using the JWE Compact Serialization, these components are 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, these components also include 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, with each value being separated from the next by a single period ('.') character, resulting in exactly four delimiting period characters being used. The JWE JSON Serialization is described in Section 7.2.

2. Base64url decode 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, following the restriction that no line breaks, white space, or other additional characters have been used.
3. Verify that the octet sequence resulting from decoding the encoded JWE Protected Header is a UTF-8 encoded representation of a completely valid JSON object conforming to RFC 7159 [RFC7159]; let the JWE Protected Header be this JSON object.
4. If using the JWE Compact Serialization, let the JOSE Header be the JWE Protected Header. Otherwise, when using the JWE JSON Serialization, let the JOSE 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. During this step, verify that the resulting JOSE Header does 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 object values that together comprise the JOSE Header.
5. Verify that the implementation understands and can process all fields that it is required to support, whether required by this specification, by the algorithms being used, or by the "crit" Header Parameter value, and that the values of those parameters are also understood and supported.
6. Determine the Key Management Mode employed by the algorithm specified by the "alg" (algorithm) Header Parameter.
7. Verify that the JWE uses a key known to the recipient.

8. 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.
9. 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. Also, see Section 11.5 for security considerations on mitigating timing attacks.
10. When Direct Key Agreement or Direct Encryption are employed, verify that the JWE Encrypted Key value is empty octet sequence.
11. When Direct Encryption is employed, let the Content Encryption Key (CEK) be the shared symmetric key.
12. Record whether the CEK could be successfully determined for this recipient or not.
13. If the JWE JSON Serialization is being used, repeat this process (steps 4-12) for each recipient contained in the representation.
14. Compute the Encoded Protected 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(Encoded Protected Header)`. However if a JWE AAD value is present (which can only be the case when using the JWE JSON Serialization), instead let the Additional Authenticated Data encryption parameter be `ASCII(Encoded Protected Header || '.' || BASE64URL(JWE AAD))`.
16. 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 validating 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.

17. If a "zip" parameter was included, uncompress the decrypted plaintext using the specified compression algorithm.
18. If there was no recipient for which all of the decryption steps succeeded, then the JWE MUST be considered invalid. Otherwise, output the Plaintext. In the JWE JSON Serialization case, also return a result to the application indicating for which of the recipients the decryption succeeded and failed.

Finally, note that it is an application decision which algorithms may be used in a given context. Even if a JWE can be successfully decrypted, unless the algorithms used in the JWE are acceptable to the application, it SHOULD consider the JWE to be invalid.

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

JWEs use one of two serializations: the JWE Compact Serialization or the JWE JSON Serialization. Applications using this specification need to specify what serialization and serialization features are used for that application. For instance, applications might specify that only the JWE JSON Serialization is used, that only JWE JSON Serialization support for a single recipient is used, or that support for multiple recipients is used. JWE implementations only need to implement the features needed for the applications they are designed to support.

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 and it provides no syntax to represent JWE Shared Unprotected Header, JWE Per-Recipient Unprotected Header, or JWE AAD values.

7.2. JWE JSON Serialization

The JWE JSON Serialization represents encrypted content as a JSON object. This representation is neither optimized for compactness nor URL-safe.

Two closely related syntaxes are defined for the JWE JSON Serialization: a fully general syntax, with which content can be encrypted to more than one recipient, and a flattened syntax, which is optimized for the single recipient case.

7.2.1. General JWE JSON Serialization Syntax

The following members are defined for use in top-level JSON objects used for the fully general JWE JSON Serialization syntax:

protected

The "protected" member MUST be present and contain the value `BASE64URL(UTF8(JWE Protected Header))` when the JWE Protected Header value is non-empty; otherwise, it MUST be absent. These Header Parameter values are integrity protected.

unprotected

The "unprotected" member MUST be present and contain the value JWE Shared Unprotected Header when the JWE Shared Unprotected Header value is non-empty; otherwise, it MUST be absent. This value is represented as an unencoded JSON object, rather than as a string. These Header Parameter values are not integrity protected.

iv

The "iv" member MUST be present and contain the value `BASE64URL(JWE Initialization Vector)` when the JWE Initialization Vector value is non-empty; otherwise, it MUST be absent.

aad

The "aad" member MUST be present and contain the value `BASE64URL(JWE AAD)` when the JWE AAD value is non-empty; otherwise, it MUST be absent. A JWE AAD value can be included to

supply a base64url encoded value to be integrity protected but not encrypted.

ciphertext

The "ciphertext" member MUST be present and contain the value BASE64URL(JWE Ciphertext).

tag

The "tag" member MUST be present and contain the value BASE64URL(JWE Authentication Tag) when the JWE Authentication Tag value is non-empty; otherwise, it MUST be absent.

recipients

The "recipients" member value MUST be an array of JSON objects. Each object contains information specific to a single recipient. This member MUST be present with exactly one array element per recipient, even if some or all of the array element values are 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 following members are defined for use in the JSON objects that are elements of the "recipients" array:

header

The "header" member MUST be present and contain the value JWE Per-Recipient Unprotected Header when the JWE Per-Recipient Unprotected Header value is non-empty; otherwise, it MUST be absent. This value is represented as an unencoded JSON object, rather than as a string. These Header Parameter values are not integrity protected.

encrypted_key

The "encrypted_key" member MUST be present and contain the value BASE64URL(JWE Encrypted Key) when the JWE Encrypted Key value is non-empty; otherwise, it MUST be absent.

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.

Additional members can be present in both the JSON objects defined above; if not understood by implementations encountering them, they MUST be ignored.

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.

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 represented in the "protected" member, (2) the JWE Shared Unprotected Header represented in the "unprotected" member, and (3) the JWE Per-Recipient Unprotected Header represented in the "header" member of the recipient's array element. The union of these sets of Header Parameters comprises the JOSE Header. The Header Parameter names in the three locations MUST be disjoint.

Each JWE Encrypted Key value is computed using the parameters of the corresponding JOSE 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, when present, 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 algorithm) Header Parameter value in the JOSE Header for each recipient and any parameters of that algorithm MUST be the same.

In summary, the syntax of a JWE using the general 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>"
}
```

```
}
```

See Appendix A.4 for an example JWE using the general JWE JSON Serialization syntax.

7.2.2. Flattened JWE JSON Serialization Syntax

The flattened JWE JSON Serialization syntax is based upon the general syntax, but flattens it, optimizing it for the single recipient case. It flattens it by removing the "recipients" member and instead placing those members defined for use in the "recipients" array (the "header" and "encrypted_key" members) in the top-level JSON object (at the same level as the "ciphertext" member).

The "recipients" member MUST NOT be present when using this syntax. Other than this syntax difference, JWE JSON Serialization objects using the flattened syntax are processed identically to those using the general syntax.

In summary, the syntax of a JWE using the flattened JWE JSON Serialization is as follows:

```
{
  "protected": "<integrity-protected header contents>",
  "unprotected": "<non-integrity-protected header contents>",
  "header": "<more non-integrity-protected header contents>",
  "encrypted_key": "<encrypted key contents>",
  "aad": "<additional authenticated data contents>",
  "iv": "<initialization vector contents>",
  "ciphertext": "<ciphertext contents>",
  "tag": "<authentication tag contents>"
}
```

Note that when using the flattened syntax, just as when using the general syntax, any unprotected Header Parameter values can reside in either the "unprotected" member or the "header" member, or in both.

See Appendix A.5 for an example JWE using the flattened JWE JSON Serialization syntax.

8. TLS Requirements

The TLS requirements for this specification are the same as those defined in Section 8 of [JWS].

9. Distinguishing between JWS and JWE Objects

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

- 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 "payload" member and JWEs do not. JWEs have a "ciphertext" member and JWSs do not.
- o The JOSE Header for a JWS can be distinguished from the JOSE Header for a JWE 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. (Extracting the "alg" value to examine is straightforward when using the JWS Compact Serialization or the JWE Compact Serialization and may be more difficult when using the JWS JSON Serialization or the JWE JSON Serialization.)
- o The JOSE Header for a JWS can also be distinguished from the JOSE Header for a JWE by determining whether an "enc" (encryption algorithm) member exists. If the "enc" member exists, it is a JWE; otherwise, it is a JWS.

10. IANA Considerations

10.1. JSON Web Signature and Encryption Header Parameters 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].

10.1.1. Registry Contents

- o Header Parameter Name: "alg"
- o Header Parameter Description: Algorithm

- 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 Description: Encryption Algorithm
- 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 Description: Compression Algorithm
- 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 Description: JWK Set URL
- 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 Description: JSON Web Key
- 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: "kid"
- o Header Parameter Description: Key ID
- 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: "x5u"
- o Header Parameter Description: X.509 URL
- 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 Description: X.509 Certificate Chain
- 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: "x5t"
- o Header Parameter Description: X.509 Certificate SHA-1 Thumbprint
- 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: "x5t#S256"
- o Header Parameter Description: X.509 Certificate SHA-256 Thumbprint
- 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: "typ"
- o Header Parameter Description: Type
- 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: "cty"
- o Header Parameter Description: Content Type
- o Header Parameter Usage Location(s): JWE
- o Change Controller: IESG
- o Specification Document(s): Section 4.1.12 of [[this document]]

- o Header Parameter Name: "crit"
- o Header Parameter Description: Critical
- o Header Parameter Usage Location(s): JWE
- o Change Controller: IESG
- o Specification Document(s): Section 4.1.13 of [[this document]]

11. Security Considerations

All of the security issues that are pertinent to any cryptographic application must be addressed by JWS/JWE/JWK agents. Among these issues are protecting the user's asymmetric private and symmetric secret keys and employing countermeasures to various attacks.

All the security considerations in the JWS specification also apply to this specification. Likewise, all the security considerations in XML Encryption 1.1 [W3C.REC-xmlenc-core1-20130411] also apply, other than those that are XML specific.

11.1. Key Entropy and Random Values

See Section 10.1 of [JWS] for security considerations on key entropy and random values. In addition to the uses of random values listed there, note that random values are also used for content encryption

keys (CEKs) and initialization vectors (IVs) when performing encryption.

11.2. Key Protection

See Section 10.2 of [JWS] for security considerations on key protection. In addition to the keys listed there that must be protected, implementations performing encryption must protect the key encryption key and the content encryption key. Compromise of the key encryption key may result in the disclosure of all contents protected with that key. Similarly, compromise of the content encryption key may result in disclosure of the associated encrypted content.

11.3. Using Matching Algorithm Strengths

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. If the key encryption and content encryption algorithms are different, the effective security is determined by the weaker of the two algorithms.

Also, see RFC 3766 [RFC3766] for information on determining strengths for public keys used for exchanging symmetric keys.

11.4. Adaptive Chosen-Ciphertext Attacks

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.

Additionally, this type of attack can be prevented by restricting the use of a key to a limited set of algorithms -- usually one. This means, for instance, that if the key is marked as being for "RSA-OAEP" only, any attempt to decrypt a message using the "RSA1_5" algorithm with that key should fail immediately due to invalid use of the key.

11.5. Timing Attacks

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 recipient substitute a randomly generated CEK and proceed to the next step, to mitigate timing attacks.

12. References

12.1. Normative References

- [JWA] Jones, M., "JSON Web Algorithms (JWA)", draft-ietf-jose-json-web-algorithms (work in progress), January 2015.
- [JWK] Jones, M., "JSON Web Key (JWK)", draft-ietf-jose-json-web-key (work in progress), January 2015.
- [JWS] Jones, M., Bradley, J., and N. Sakimura, "JSON Web Signature (JWS)", draft-ietf-jose-json-web-signature (work in progress), January 2015.
- [RFC1951] Deutsch, P., "DEFLATE Compressed Data Format Specification version 1.3", RFC 1951, May 1996.
- [RFC20] Cerf, V., "ASCII format for Network Interchange", RFC 20, October 1969.
- [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.
- [RFC4949] Shirey, R., "Internet Security Glossary, Version 2", RFC 4949, August 2007.
- [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.
- [RFC7159] Bray, T., "The JavaScript Object Notation (JSON) Data Interchange Format", RFC 7159, March 2014.

[UNICODE] The Unicode Consortium, "The Unicode Standard", 1991-,
<<http://www.unicode.org/versions/latest/>>.

12.2. Informative References

- [AES] National Institute of Standards and Technology (NIST),
"Advanced Encryption Standard (AES)", FIPS PUB 197,
November 2001.
- [I-D.mcgregw-aead-aes-cbc-hmac-sha2]
McGrew, D., Foley, J., and K. Paterson, "Authenticated
Encryption with AES-CBC and HMAC-SHA",
draft-mcgregw-aead-aes-cbc-hmac-sha2-05 (work in progress),
July 2014.
- [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.
- [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.
- [RFC3218] Rescorla, E., "Preventing the Million Message Attack on
Cryptographic Message Syntax", RFC 3218, January 2002.
- [RFC3447] Jonsson, J. and B. Kaliski, "Public-Key Cryptography
Standards (PKCS) #1: RSA Cryptography Specifications
Version 2.1", RFC 3447, February 2003.
- [RFC3766] Orman, H. and P. Hoffman, "Determining Strengths For
Public Keys Used For Exchanging Symmetric Keys", BCP 86,
RFC 3766, April 2004.
- [RFC4086] Eastlake, D., Schiller, J., and S. Crocker, "Randomness
Requirements for Security", BCP 106, RFC 4086, June 2005.
- [RFC5652] Housley, R., "Cryptographic Message Syntax (CMS)", STD 70,
RFC 5652, September 2009.
- [W3C.REC-xmlenc-core1-20130411]
Eastlake, D., Reagle, J., Hirsch, F., and T. Roessler,

"XML Encryption Syntax and Processing Version 1.1", World Wide Web Consortium Recommendation REC-xmlenc-core1-20130411, April 2013, <<http://www.w3.org/TR/2013/REC-xmlenc-core1-20130411/>>.

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 (using JSON array notation) 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. JOSE 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.
- o Authenticated encryption is performed on the Plaintext using the AES GCM algorithm with a 256 bit key to produce the Ciphertext and the Authentication Tag.

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

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

```
eyJhbGciOiJSU0EtT0FFUCIsImVuYyI6IkeYNTZHQ00ifQ
```

A.1.2. Content Encryption Key (CEK)

Generate a 256 bit random Content Encryption Key (CEK). In this example, the value (using JSON array notation) 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 within values for display purposes only):

```
{ "kty": "RSA",
  "n": "oahUIoWw0K0usKNuOR6H4wkf4oBUXHTxRvgb48E-BVvxkeDNjbC4he8rUW
cJoZmds2h7M70imEVhRU5djINXtql1XI4DFqcI1Dgjt9LewND8MW2Krf3S
psk_ZkoFnilakGygTwpZ3uesH-PFABNIUYp0iN15dsQRkgr0vEhxN92i2a
sbOenSZeyaxziK72UwxrrKoExv6kc5twXTq4h-QChL0ln0_mtUZwfsRaMS
tPs6mS6XrgxnbWhojf663tuEQueGC-FCMfra36C9knDFGzKsNa7LZK2dj
YgyD3JR_MB_4NUJW_TqOQtWYbxevoJArm-L5StowjzGy-_bq6Gw",
  "e": "AQAB",
  "d": "kLdtIj6GbDks_ApCSTYQtelcNttlKiOyPzMrXHeI-yk1F7-kpDxY4-WY5N
WV5KntaEeXS1j82E375xxhWMHXyvJYecPT9fpwR_M9gV8n9Hrh2anTpTD9
3Dt62ypW3yDsJzBnTnrYuliwWRgBKREYY46qAZIrA2xAwnm2X7uGR1hghk
qDp0Vqj3kbSCz1XyfcS6_LehBwtXHIyh8Ripy40p24moOAbgxVw3rxT_vl
t3Uve4WO3JkJOzlpUf-KTVI2Ptgm-dARxTETE-id-4OJr0h-K-VFs3VSnd
VTIznSxfyrj8ILL6MG_Uv8YA7VILSB3lOW085-4qe3DzgrTjgyQ",
  "p": "1r52Xk46c-LsfB5P442p7atdPUrxQSy4mti_tZI3Mgf2EuFVbUoDBvaRQ-
SWxkbbkmoEzL7JXroSBjSrK3YIQgYdMgyAEPTPjXv_hI2_1eTSPVZfzL0lf
fNn03IXqWF5MDFuoUYE0hzb2vhrln_rKrbfDIwUbTrjjgieRbwC6Cl0",
  "q": "wLb35x7hmQWZsWJmB_vle87ihgZ19S8lBEROLIsZG4ayZVe9Hi9gDVCObm
UDdaDYVTSNx_8Fyw1YYa9XGrGnDew00J28cRUoeBB_jKiloma0Orv1T9aX
IWxKwd4gvxFImOWr3QRL9KEBRzk2RatUBnmDZJTIAfwTs0g68UZHVtc",
  "dp": "ZK-YwE7diUh0qRltR7w8WHTolDx3MZ_OTowiFvgfeQ3SiresXjm9gZ5KL
hMXvo-uz-KUJWDxS5pFQ_M0evdoldKiRTjVw_x4NyqyXPM5nULPkcpU827
rnpZzAJKpdhWAgqrXGKAECQH0Xt4taznjnd_zVpAmZZq60WPMBMfKcuE",
  "dq": "Dq0gfgJ1DdFGXiLvQEZnuKEN0UumsJBxkjydc3j4ZYdBiMRAY86x0vHCj
ywcMlYYg4yoC4YZa9hNVcsjqA3FeiL19rk8g6Qn29Tt0cj8qqyFpz9vNDB
UfCAiJVeESOjJDZPYHdHY8v1b-o-Z2X5tvLx-TCekf7oxyeKDUqKWjis",
  "qi": "VIMpMYbPf47dTlw_zDUXfPimsSegnMOAlzTaX7aGk_8urY6R8-ZW1FxU7
AlWAlWYbqq6t16Vfd7hQd0y6f1UK4SlOydb61lgwanOsXGOAOv82cHq0E3
eL4HrtZkUuKvnPrMnsUUF1fUdybVzxyjz9JF_XyaY14ardLSjf4L_FNY"
}
```

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

Perform authenticated encryption on the Plaintext with the AES GCM algorithm 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:

```

eyJhbGciOiJSU0EtT0FFUCIsImVuYyI6IkJkYyNTZHQ00ifQ.
OKOawDo13gRp2ojaHV7LFpZcgV7T6DVZKTyKOMTYUmKoTCVJRgckCL9kiMT03JGe
ipsEdY3mx_etLbbWSrFr05kLzcSr4qKAq7YN7e9jwQRb23nfa6c9d-StnImGyFDb
Sv04uVuxIp5ZmslgNxKKK2Da14B8S4rzVRltdYwam_lDp5XnZAYpQdb76FdIKLaV
mqgfwX7XWRxv2322i-vDxRfqNzo_tETKzpVLzfwiwQyeyPGLBIO56YJ7eObdv0je8
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 (using JSON array notation) 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. JOSE 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.
- o Authenticated encryption is performed on the Plaintext using the AES_128_CBC_HMAC_SHA_256 algorithm to produce the Ciphertext and the Authentication Tag.

```
{"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 within values 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  
    nq8cxjDTLZQ6Fz7jSjR1Klop-YKAUHc9GsEofQqYruPhzSA-QgajZGPbE_  
    0ZaVDJHfyd7UUBUKunFMScbflYAAOYJqVIVwaYR5zWEEceUjNnTNo_CVSj  
    -VvXLO5VZfCUAVLgW4dpf1SrtZjSt34YLSRarSb127reG_DUwg9Ch-KyvJ  
    TlSkHgUWRVGcyly7uvVGRSDwsXypdrNinPA4j1hoNdizK2zF2CWQ",  
  "p": "9gY2w6I6S6L0juEKsbeDAwpd9WMfgqFoeA9vEyEUuk4kLwBKcoelx4HG68  
    ik918hdDSE9vDQScC3xXHOAFOPJ8R9EeIABTi1VwBYnbTp87X-xcPWLEP  
    krdoUKW60tgs1aNd_Nnc9LEVVPMS390zbFxt8TN_biaBgelNgbC95sM",  
  "q": "uKlCKvKv_ZJMVcdIs5vVSU_6cPtYI1ljWytExV_skstvRSNi9r66jdd9-y  
    BhVfuG4shsp2j7rGnIio90lRBeHo6TPKWVvykPuliYhQXwljIABfw-MVsN  
    -3bQ76WLDt2SDxsHs7q7zPyUyHXmps7ycZ5c72wGkUwNOjYelmkiNS0",  
  "dp": "w0kZbV63cVRvVX6yk3C8cMxo2qCM4Y8nsq1l1mMSYhG4EcL6FWbX5h9yuv  
    ngs4iLEFk6eALoUS4vIWEwcL4txw9LsWH_zKI-hwoReoP77cOdSL4AVcra  
    Hawlkpyd2TWjE5evgbhWtOxnZee3cXJBkAi64Ik6jZxbvk-RR3pEhnCs",  
  "dq": "o_8Vl4SezckO6CNLKS_btPdFiO9_kClDsUtd2LafIIVeMZ7jnlGus_Ff  
    7B7IVx3p5KuBGOVF8L-qifLb6nQnLySGHDh132NDioZkhH7mI7hPG-PYE_  
    odApKdnqECHWw0J-F0JWnUd6D2B_1TvF9mXA2Qx-iGYn8OVVlBsmp6qU",  
  "qi": "eNho5yRBEBxhGBtQRww9QirZsB66TrfFrEG_CcteIlaCneT0ELGhYlRlC  
    tUkTRclIfuEPmNsNDPbLoLqqCVznFbvdB7x-Tl-m0l_eFTj2KiqwGqE9PZ  
    B9nNTwMVvH3VRRSLWACvPnSiwP8N5Usy-WRXS-V7TbpxIhvepTfE0NNO"  
}
```

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_NPVaXsGMOLOmwvclGyqlIKOKlnN94nHPoltGRhWhw7Zx0-kFm
lNJn8LE9XShH59_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

Perform authenticated encryption on the Plaintext with the AES_128_CBC_HMAC_SHA_256 algorithm 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-YxzZIIItRzC5hlRirb6Y5Cl_p-ko3YvkkysZIF
NPccxRU7qvelWYPxqbb2Yw8kZqa2rMWI5ng8OtvzlV7elprCbuPhcCdZ6XDP0_F8
rkXds2vE4X-ncOIM8hAYHHi29NX0mcKiRaD0-D-ljQTP-cFPgwCp6X-nZZd9OHBv
-B3oWh2TbqmScqXMR4gp_A.
AxY8DCtDaGlsbGljb3RoZQ.
KDlTtXchhZTGufMYmOYGS4HffxPSUrfmqCHXaI9wOGY.
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_128_CBC_HMAC_SHA_256 for content encryption. The representation of this plaintext (using JSON array notation) 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. JOSE 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.
- o Authenticated encryption is performed on the Plaintext using the AES_128_CBC_HMAC_SHA_256 algorithm to produce the Ciphertext and the Authentication Tag.

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

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

```
eyJhbGciOiJBMTJ8KWUiLCJencIjoiA128CBC-HS256In0
```

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": "GawggguFyGrWKav7AX4VKUg"  
}
```

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

Perform authenticated encryption on the Plaintext with the `AES_128_CBC_HMAC_SHA_256` algorithm 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.
6KB707dM9YTIGhtLvtgWQ8mKwboJW3of9locizkDTHzBC2IlrTloOQ.
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 General JWE JSON Serialization

This section contains an example using the general JWE JSON Serialization syntax. 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), JWE 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

Authenticated encryption is performed on the Plaintext 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 JOSE Header Values

Combining the per-recipient, protected, and unprotected header values supplied, the JOSE 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

Perform authenticated encryption on the Plaintext with the AES_128_CBC_HMAC_SHA_256 algorithm 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 JWE JSON Serialization for these values is as follows (with line breaks within values for display purposes only):

```
{
  "protected":
    "eyJlbmMiOiJBMTI4Q0JDLUhTMjU2In0",
  "unprotected":
    { "jku": "https://server.example.com/keys.jwks" },
  "recipients": [
    { "header":
        { "alg": "RSA1_5", "kid": "2011-04-29" },
      "encrypted_key":
        "UGhIOguC7IuEvf_NPVaXsGMOmLomwvc1GyqlIKOKlnN94nHPoltGRhWhw7Zx0-
        kFmlNJn8LE9XShH59_i8J0PH5ZZyNfGy2xGdULU7sHNF6Gp2vPLgNZ__deLKx
        GHZ7PcHALUzoOegEI-8E66jX2E4zyJKx-YxzZIIItRzC5hlRirb6Y5Cl_p-ko3
        YvkkysZIFNPccxRU7qvelWYPxqbb2Yw8kZqa2rMWI5ng8Otvz1V7elprCbuPh
        cCdZ6XDP0_F8rkXds2vE4X-ncOIM8hAYHHi29NX0mcKiRaD0-D-1jQTP-cFPg
        wCp6X-nZZd9OHBv-B3oWh2TbqmScqXMR4gp_A" },
    { "header":
        { "alg": "A128KW", "kid": "7" },
      "encrypted_key":
        "6KB707dM9YTIgHtLvtgWQ8mKwboJW3of9locizkDTHzBC2IlrT1oOQ" } ] ,
  "iv":
    "Axy8DcTdaGlsbGljb3RoZQ",
  "ciphertext":
    "KDlTtXchhZTGufMYmOYGS4HffxPSUrfmqCHXaI9wOGY",
  "tag":
    "Mz-VPPyU4RlcuYv1IwIvzw"
}
```

A.5. Example JWE using Flattened JWE JSON Serialization

This section contains an example using the flattened JWE JSON Serialization syntax. This example demonstrates the capability for encrypting the plaintext to a single recipient in a flattened JSON structure.

The values in this example are the same as those for the second recipient of the previous example in Appendix A.4.

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

```
{
  "protected":
    "eyJlbmMiOiJBMTI4Q0JDLUhTMjU2In0",
  "unprotected":
    { "jku": "https://server.example.com/keys.jwks" },
  "header":
    { "alg": "A128KW", "kid": "7" },
  "encrypted_key":
    "6KB707dM9YTIgHtLvtgWQ8mKwboJW3of9locizkDTHzBC2IlrT1oOQ",
  "iv":
    "AxY8DCtDaGlsbGljb3RoZQ",
  "ciphertext":
    "KDlTtXchhZTGufMYmOYGS4HffxPSUrfmqCHXaI9wOGY",
  "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 5.2 and 5.2.3 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 #7 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 (using JSON array notation) 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 #7 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.REC-xmlenc-core1-20130411] and RFC 5652 [RFC5652] as possible, while utilizing simple, compact JSON-based data structures.

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

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, Alissa Cooper, Breno de Medeiros, Stephen Farrell, Dick Hardt, Jeff Hodges, Russ Housley, Edmund Jay, Scott Kelly, Stephen Kent, Barry Leiba, James Manger, Matt Miller, Kathleen Moriarty, Tony Nadalin, Hideki Nara, Axel Nennker, Ray Polk, Emmanuel Raviart, Eric Rescorla, Pete Resnick, Nat Sakimura, Jim Schaad, Hannes Tschofenig, and Sean Turner.

Jim Schaad and Karen O'Donoghue chaired the JOSE working group and Sean Turner, Stephen Farrell, and Kathleen Moriarty 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]]

-40

- o Clarified the definitions of UTF8(String) and ASCII(String).

-39

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

-38

- o Replaced uses of the phrases "JWS object" and "JWE object" with "JWS" and "JWE".
- o Added member names to the JWE JSON Serialization Overview.
- o Applied other minor editorial improvements.

-37

- o Restricted algorithm names to using only ASCII characters.
- o When describing actions taken as a result of validation failures, changed statements about rejecting the JWE to statements about considering the JWE to be invalid.
- o Added the CRT parameter values to example RSA private key representations.

-36

- o Defined a flattened JWE JSON Serialization syntax, which is optimized for the single recipient case.
- o Clarified where white space and line breaks may occur in JSON objects by referencing Section 2 of RFC 7159.

-35

- o Addressed AppsDir reviews by Ray Polk.

-34

- o Addressed IESG review comments by Barry Leiba, Alissa Cooper, Pete Resnick, Stephen Farrell, and Richard Barnes.

-33

- o Noted that certificate thumbprints are also sometimes known as certificate fingerprints.
- o Changed to use the term "authenticated encryption" instead of "encryption", where appropriate.
- o Acknowledged additional contributors.

-32

- o Addressed Gen-ART review comments by Russ Housley.
- o Addressed secdir review comments by Scott Kelly, Tero Kivinen, and Stephen Kent.

-31

- o Updated the reference to draft-mcgrew-aead-aes-cbc-hmac-sha2.

-30

- o Added subsection headings within the Overview section for the two serializations.
- o Added references and cleaned up the reference syntax in a few places.
- o Applied minor wording changes to the Security Considerations section and made other local editorial improvements.

-29

- o Replaced the terms JWS Header, JWE Header, and JWT Header with a single JOSE Header term defined in the JWS specification. This also enabled a single Header Parameter definition to be used and reduced other areas of duplication between specifications.

-28

- o Specified the use of PKCS #7 padding with AES CBC, rather than PKCS #5. (PKCS #7 is a superset of PKCS #5, and is appropriate for the 16 octet blocks used by AES CBC.)
- o Revised the introduction to the Security Considerations section. Also moved a security consideration item here from the JWA draft.

-27

- o Described additional security considerations.
- o Added the "x5t#S256" (X.509 Certificate SHA-256 Thumbprint) header parameter.

-26

- o Noted that octet sequences are depicted using JSON array notation.
- o Updated references, including to W3C specifications.

-25

- o Corrected two external section number references that had changed.
- o Corrected a typo in an algorithm name in the prose of an example.

-24

- o Corrected complete JSON Serialization example.

- o Replaced uses of the term "associated data" wherever it was used to refer to a data value with "additional authenticated data", since both terms were being used as synonyms, causing confusion.
- o Updated the JSON reference to RFC 7159.
- o Thanked Eric Rescorla for helping to author of most of the drafts of this specification and removed him from the current author list.

-23

- o Corrected a use of the word "payload" to "plaintext".

-22

- o Corrected RFC 2119 terminology usage.
- o Replaced references to draft-ietf-json-rfc4627bis with RFC 7158.

-21

- o Changed some references from being normative to informative, addressing issue #90.
- o Applied review comments to the JSON Serialization section, addressing issue #178.

-20

- o Made terminology definitions more consistent, addressing issue #165.
- o Restructured the JSON Serialization section to call out the parameters used in hanging lists, addressing issue #178.
- o Replaced references to RFC 4627 with draft-ietf-json-rfc4627bis, addressing issue #90.

-19

- o Reordered the key selection parameters.

-18

- o Updated the mandatory-to-implement (MTI) language to say that applications using this specification need to specify what serialization and serialization features are used for that

application, addressing issue #176.

- o Changes to address editorial and minor issues #89, #135, #165, #174, #175, #177, #179, and #180.
- o Used Header Parameter Description registry field.

-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.mcgregor-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 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/>

Joe Hildebrand
Cisco Systems, Inc.

Email: jhildebr@cisco.com

JOSE Working Group
Internet-Draft
Intended status: Standards Track
Expires: July 20, 2015

M. Jones
Microsoft
January 16, 2015

JSON Web Key (JWK)
draft-ietf-jose-json-web-key-41

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 that represents 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 July 20, 2015.

Copyright Notice

Copyright (c) 2015 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. Example JWK	5
4. JSON Web Key (JWK) Format	5
4.1. "kty" (Key Type) Parameter	6
4.2. "use" (Public Key Use) Parameter	6
4.3. "key_ops" (Key Operations) Parameter	7
4.4. "alg" (Algorithm) Parameter	8
4.5. "kid" (Key ID) Parameter	8
4.6. "x5u" (X.509 URL) Parameter	8
4.7. "x5c" (X.509 Certificate Chain) Parameter	9
4.8. "x5t" (X.509 Certificate SHA-1 Thumbprint) Parameter	9
4.9. "x5t#S256" (X.509 Certificate SHA-256 Thumbprint) Parameter	10
5. JSON Web Key Set (JWK Set) Format	10
5.1. "keys" Parameter	11
6. String Comparison Rules	11
7. Encrypted JWK and Encrypted JWK Set Formats	11
8. IANA Considerations	12
8.1. JSON Web Key Parameters Registry	13
8.1.1. Registration Template	13
8.1.2. Initial Registry Contents	14
8.2. JSON Web Key Use Registry	15
8.2.1. Registration Template	16
8.2.2. Initial Registry Contents	16
8.3. JSON Web Key Operations Registry	16
8.3.1. Registration Template	17
8.3.2. Initial Registry Contents	17
8.4. JSON Web Key Set Parameters Registry	18
8.4.1. Registration Template	18
8.4.2. Initial Registry Contents	19
8.5. Media Type Registration	19
8.5.1. Registry Contents	19
9. Security Considerations	20
9.1. Key Provenance and Trust	20
9.2. Preventing Disclosure of Non-Public Key Information	21
9.3. RSA Private Key Representations and Blinding	21
9.4. Key Entropy and Random Values	22
10. References	22
10.1. Normative References	22
10.2. Informative References	24

Appendix A. Example JSON Web Key Sets	25
A.1. Example Public Keys	25
A.2. Example Private Keys	25
A.3. Example Symmetric Keys	27
Appendix B. Example Use of "x5c" (X.509 Certificate Chain) Parameter	27
Appendix C. Example Encrypted RSA Private Key	28
C.1. Plaintext RSA Private Key	29
C.2. JOSE Header	32
C.3. Content Encryption Key (CEK)	32
C.4. Key Derivation	33
C.5. Key Encryption	33
C.6. Initialization Vector	33
C.7. Additional Authenticated Data	34
C.8. Content Encryption	34
C.9. Complete Representation	37
Appendix D. Acknowledgements	39
Appendix E. Document History	39
Author's Address	47

1. Introduction

A JSON Web Key (JWK) is a JavaScript Object Notation (JSON) [RFC7159] data structure that represents a cryptographic key. This specification also defines a JSON Web Key Set (JWK Set) JSON data structure that represents 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 new kinds of certificate chains, representing new kinds of certified keys, or 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", "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 of [JWS].

UTF8(String) denotes the octets of the UTF-8 [RFC3629] representation of String, where String is a sequence of zero or more Unicode [UNICODE] characters.

ASCII(String) denotes the octets of the ASCII [RFC20] representation of String, where String is a sequence of zero or more ASCII characters.

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)", "Base64url Encoding", "Collision-Resistant Name",

"Header Parameter", and "JOSE Header".

These terms defined by the JSON Web Encryption (JWE) [JWE] specification are incorporated into this specification: "JSON Web Encryption (JWE)", "Additional Authenticated Data (AAD)", "JWE Authentication Tag", "JWE Ciphertext", "JWE Compact Serialization", "JWE Encrypted Key", "JWE Initialization Vector", and "JWE Protected Header".

These terms defined by the Internet Security Glossary, Version 2 [RFC4949] are incorporated into this specification: "Ciphertext", "Digital Signature", "Message Authentication Code (MAC)", and "Plaintext".

These terms are defined by this specification:

JSON Web Key (JWK)

A JSON object that represents a cryptographic key. The members of the object represent properties of the key, including its value.

JSON Web Key Set (JWK Set)

A JSON object that represents a set of JWKs. The JSON object MUST have a "keys" member, which is an array of JWKs.

3. Example JWK

This section provides an example of a JWK. The following example JWK declares that the key is an Elliptic Curve [DSS] key, it is used with the P-256 Elliptic Curve, and its x and y coordinates are the base64url encoded values shown. A key identifier is also provided for the key.

```
{ "kty": "EC",  
  "crv": "P-256",  
  "x": "f83OJ3D2xF1Bg8vub9tLe1gHMzV76e8Tus9uPHvRVEU",  
  "y": "x_FEzRu9m36HLN_tue659LNpXW6pCyStikYjKIWI5a0",  
  "kid": "Public key used in JWS A.3 example"  
}
```

Additional example JWK values can be found in Appendix A.

4. JSON Web Key (JWK) Format

A JSON Web Key (JWK) is a JSON object that represents a cryptographic key. The members of the object represent properties of the key, including its value. This JSON object MAY contain white space and/or

line breaks before or after any JSON values or structural characters, in accordance with Section 2 of RFC 7159 [RFC7159]. This document defines the key parameters that are not algorithm specific, and thus common to many keys.

In addition to the common parameters, each JWK will have members that are key type-specific. These members represent the parameters of the key. Section 6 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; JWK parsers 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 can 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 keys types need not be distinct. Any new member name should either be registered in the IANA JSON Web Key Parameters registry defined in Section 8.1 or be a value that contains a Collision-Resistant Name.

4.1. "kty" (Key Type) Parameter

The "kty" (key type) member identifies the cryptographic algorithm family used with the key, such as "RSA" or "EC". "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. This member MUST be present in a JWK.

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 6.1 of the JSON Web Algorithms (JWA) [JWA] specification.

The key type definitions include specification of the members to be used for those key types. Additional members used with "kty" values can also be found in the IANA JSON Web Key Parameters registry defined in Section 8.1.

4.2. "use" (Public Key Use) Parameter

The "use" (public key use) member identifies the intended use of the public key. The "use" parameter is employed to indicate whether a public key is used for encrypting data or verifying the signature on data.

Values defined by this specification are:

- o "sig" (signature)
- o "enc" (encryption)

Other values MAY be used. The "use" value is a case-sensitive string. Use of the "use" member is OPTIONAL, unless the application requires its presence.

When a key is used to wrap another key and a Public Key Use designation for the first key is desired, the "enc" (encryption) key use value is used, since key wrapping is a kind of encryption. The "enc" value is also be used for public keys used for key agreement operations.

Additional Public Key Use values can be registered in the IANA JSON Web Key Use registry defined in Section 8.2. Registering any extension values used is highly recommended when this specification is used in open environments, in which multiple organizations need to have a common understanding of any extensions used. However, unregistered extension values can be used in closed environments, in which the producing and consuming organization will always be the same.

4.3. "key_ops" (Key Operations) Parameter

The "key_ops" (key operations) member identifies the operation(s) that the key is intended to be used for. The "key_ops" parameter is intended for use cases in which public, private, or symmetric keys may be present.

Its value is an array of key operation values. Values defined by this specification are:

- o "sign" (compute digital signature or MAC)
- o "verify" (verify digital signature or MAC)
- o "encrypt" (encrypt content)
- o "decrypt" (decrypt content and validate decryption, if applicable)
- o "wrapKey" (encrypt key)
- o "unwrapKey" (decrypt key and validate decryption, if applicable)
- o "deriveKey" (derive key)
- o "deriveBits" (derive bits not to be used as a key)

(Note that the "key_ops" values intentionally match the "KeyUsage" values defined in the Web Cryptography API [W3C.CR-WebCryptoAPI-20141211] specification.)

Other values MAY be used. The key operation values are case-

sensitive strings. Duplicate key operation values MUST NOT be present in the array. Use of the "key_ops" member is OPTIONAL, unless the application requires its presence.

Multiple unrelated key operations SHOULD NOT be specified for a key because of the potential vulnerabilities associated with using the same key with multiple algorithms. Thus, the combinations "sign" with "verify", "encrypt" with "decrypt", and "wrapKey" with "unwrapKey" are permitted, but other combinations SHOULD NOT be used.

Additional Key Operations values can be registered in the IANA JSON Web Key Operations registry defined in Section 8.3. The same considerations about registering extension values apply to the "key_ops" member as do for the "use" member.

The "use" and "key_ops" JWK members SHOULD NOT be used together; however, if both are used, the information they convey MUST be consistent. Applications should specify which of these members they use, if either is to be used by the application.

4.4. "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. The "alg" value is a case-sensitive ASCII string. Use of this member is OPTIONAL.

4.5. "kid" (Key ID) Parameter

The "kid" (key ID) member is used to match a specific key. This is used, for instance, to choose among a set of keys within a JWK Set during key rollover. The structure 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. (One example in which different keys might use the same "kid" value is if they have different "kty" (key type) values but are considered to be equivalent alternatives by the application using them.) The "kid" value is a case-sensitive string. Use of this member is OPTIONAL.

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

4.6. "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, with each certificate delimited as specified in Section 6.1 of RFC 4945 [RFC4945]. The key in the first certificate MUST match the 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 6 of RFC 6125 [RFC6125]. Use of this member is OPTIONAL.

While there is no requirement that optional JWK members providing key usage, algorithm, or other information be present when the "x5u" member is used, doing so may improve interoperability for applications that do not handle PKIX certificates [RFC5280]. If other members are present, the contents of those members MUST be semantically consistent with the related fields in the first certificate. For instance, if the "use" member is present, then it MUST correspond to the usage that is specified in the certificate, when it includes this information. Similarly, if the "alg" member is present, it MUST correspond to the algorithm specified in the certificate.

4.7. "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 public key represented by other members of the JWK. Use of this member is OPTIONAL.

As with the "x5u" member, optional JWK members providing key usage, algorithm, or other information MAY also be present when the "x5c" member is used. If other members are present, the contents of those members MUST be semantically consistent with the related fields in the first certificate. See the last paragraph of Section 4.6 for additional guidance on this.

4.8. "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]. Note that certificate thumbprints are also sometimes known as certificate fingerprints. The key in the certificate MUST match the public key represented by other members of the JWK. Use of this member is OPTIONAL.

As with the "x5u" member, optional JWK members providing key usage, algorithm, or other information MAY also be present when the "x5t" member is used. If other members are present, the contents of those members MUST be semantically consistent with the related fields in the referenced certificate. See the last paragraph of Section 4.6 for additional guidance on this.

4.9. "x5t#S256" (X.509 Certificate SHA-256 Thumbprint) Parameter

The "x5t#S256" (X.509 Certificate SHA-256 Thumbprint) member is a base64url encoded SHA-256 thumbprint (a.k.a. digest) of the DER encoding of an X.509 certificate [RFC5280]. Note that certificate thumbprints are also sometimes known as certificate fingerprints. The key in the certificate MUST match the public key represented by other members of the JWK. Use of this member is OPTIONAL.

As with the "x5u" member, optional JWK members providing key usage, algorithm, or other information MAY also be present when the "x5t#S256" member is used. If other members are present, the contents of those members MUST be semantically consistent with the related fields in the referenced certificate. See the last paragraph of Section 4.6 for additional guidance on this.

5. JSON Web Key Set (JWK Set) Format

A JSON Web Key Set (JWK Set) is a JSON object that represents a set of JWKs. The JSON object MUST have a "keys" member, with its value being an array of JWKs. This JSON object MAY contain white space and/or line breaks.

The member names within a JWK Set MUST be unique; JWK Set parsers 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 can 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 8.4 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, are missing required members, or for which values are out of the supported ranges.

5.1. "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.

6. String Comparison Rules

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

7. Encrypted JWK and Encrypted JWK Set Formats

Access to JWKs containing non-public key material by parties without legitimate access to the non-public information MUST be prevented. This can be accomplished by encrypting the JWK when potentially observable by such parties to prevent the disclosure of private or symmetric key values. The use of an Encrypted JWK, which is a JWE with the UTF-8 encoding of 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, in which case the "cty" value would typically be omitted.

JWK Sets containing non-public key material will also need to be encrypted under these circumstances. The use of an Encrypted JWK Set, which is a JWE with the UTF-8 encoding of 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, in which case the "cty" value would typically be omitted.

See Appendix C for an example encrypted JWK.

8. IANA Considerations

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

Values are registered on a Specification Required [RFC5226] basis after a three-week review period on the jose-reg-review@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 jose-reg-review@ietf.org mailing list for review and comment, with an appropriate subject (e.g., "Request to register JWK parameter: example").

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@ietf.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 description is clear.

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).

[[Note to the RFC Editor and IANA: Pearl Liang of ICANN had requested that the draft supply the following proposed registry description information. It is to be used for all registries established by this specification.

- o Protocol Category: JSON Object Signing and Encryption (JOSE)
- o Registry Location: <http://www.iana.org/assignments/jose>
- o Webpage Title: (same as the protocol category)
- o Registry Name: (same as the section title, but excluding the word "Registry", for example "JSON Web Key Parameters")

]]

8.1. 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 4. 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.

8.1.1. Registration Template

Parameter Name:

The name requested (e.g., "kid"). 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. However, matching names may be registered, provided that the accompanying sets of "kty" values that the Parameter Name is used with are disjoint; for the purposes of matching "kty" values, "*" matches all values.

Parameter Description:

Brief description of the parameter (e.g., "Key ID").

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. Values may not match other registered "kty" values in a case-insensitive manner when the registered Parameter Name is

the same (including when the Parameter Name matches 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.

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.

8.1.2. Initial Registry Contents

- o Parameter Name: "kty"
- o Parameter Description: Key Type
- o Used with "kty" Value(s): *
- o Parameter Information Class: Public
- o Change Controller: IESG
- o Specification Document(s): Section 4.1 of [[this document]]

- o Parameter Name: "use"
- o Parameter Description: Public Key Use
- o Used with "kty" Value(s): *
- o Parameter Information Class: Public
- o Change Controller: IESG
- o Specification Document(s): Section 4.2 of [[this document]]

- o Parameter Name: "key_ops"
- o Parameter Description: Key Operations
- o Used with "kty" Value(s): *
- o Parameter Information Class: Public
- o Change Controller: IESG
- o Specification Document(s): Section 4.3 of [[this document]]

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

- o Parameter Information Class: Public
- o Change Controller: IESG
- o Specification Document(s): Section 4.4 of [[this document]]

- o Parameter Name: "kid"
- o Parameter Description: Key ID
- o Used with "kty" Value(s): *
- o Parameter Information Class: Public
- o Change Controller: IESG
- o Specification Document(s): Section 4.5 of [[this document]]

- o Parameter Name: "x5u"
- o Parameter Description: X.509 URL
- o Used with "kty" Value(s): *
- o Parameter Information Class: Public
- o Change Controller: IESG
- o Specification Document(s): Section 4.6 of [[this document]]

- o Parameter Name: "x5c"
- o Parameter Description: X.509 Certificate Chain
- o Used with "kty" Value(s): *
- o Parameter Information Class: Public
- o Change Controller: IESG
- o Specification Document(s): Section 4.7 of [[this document]]

- o Parameter Name: "x5t"
- o Parameter Description: X.509 Certificate SHA-1 Thumbprint
- o Used with "kty" Value(s): *
- o Parameter Information Class: Public
- o Change Controller: IESG
- o Specification Document(s): Section 4.8 of [[this document]]

- o Parameter Name: "x5t#S256"
- o Parameter Description: X.509 Certificate SHA-256 Thumbprint
- o Used with "kty" Value(s): *
- o Parameter Information Class: Public
- o Change Controller: IESG
- o Specification Document(s): Section 4.9 of [[this document]]

8.2. JSON Web Key Use Registry

This specification establishes the IANA JSON Web Key Use registry for JWK "use" (public key use) member values. The registry records the public key use value and a reference to the specification that defines it. This specification registers the parameter names defined in Section 4.2.

8.2.1. Registration Template

Use Member Value:

The name requested (e.g., "sig"). 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.

Use Description:

Brief description of the use (e.g., "Digital Signature or MAC").

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.

8.2.2. Initial Registry Contents

- o Use Member Value: "sig"
- o Use Description: Digital Signature or MAC
- o Change Controller: IESG
- o Specification Document(s): Section 4.2 of [[this document]]

- o Use Member Value: "enc"
- o Use Description: Encryption
- o Change Controller: IESG
- o Specification Document(s): Section 4.2 of [[this document]]

8.3. JSON Web Key Operations Registry

This specification establishes the IANA JSON Web Key Operations registry for values of JWK "key_ops" array elements. The registry records the key operation value and a reference to the specification that defines it. This specification registers the parameter names defined in Section 4.3.

8.3.1. Registration Template

Key Operation Value:

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

Key Operation Description:

Brief description of the key operation (e.g., "Compute digital signature or MAC").

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.

8.3.2. Initial Registry Contents

- o Key Operation Value: "sign"
- o Key Operation Description: Compute digital signature or MAC
- o Change Controller: IESG
- o Specification Document(s): Section 4.3 of [[this document]]

- o Key Operation Value: "verify"
- o Key Operation Description: Verify digital signature or MAC
- o Change Controller: IESG
- o Specification Document(s): Section 4.3 of [[this document]]

- o Key Operation Value: "encrypt"
- o Key Operation Description: Encrypt content
- o Change Controller: IESG
- o Specification Document(s): Section 4.3 of [[this document]]

- o Key Operation Value: "decrypt"
- o Key Operation Description: Decrypt content and validate decryption, if applicable

- o Change Controller: IESG
- o Specification Document(s): Section 4.3 of [[this document]]
- o Key Operation Value: "wrapKey"
- o Key Operation Description: Encrypt key
- o Change Controller: IESG
- o Specification Document(s): Section 4.3 of [[this document]]
- o Key Operation Value: "unwrapKey"
- o Key Operation Description: Decrypt key and validate decryption, if applicable
- o Change Controller: IESG
- o Specification Document(s): Section 4.3 of [[this document]]
- o Key Operation Value: "deriveKey"
- o Key Operation Description: Derive key
- o Change Controller: IESG
- o Specification Document(s): Section 4.3 of [[this document]]
- o Key Operation Value: "deriveBits"
- o Key Operation Description: Derive bits not to be used as a key
- o Change Controller: IESG
- o Specification Document(s): Section 4.3 of [[this document]]

8.4. 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 5.

8.4.1. Registration Template

Parameter Name:

The name requested (e.g., "keys"). 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.

Parameter Description:

Brief description of the parameter (e.g., "Array of JWK values").

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.

8.4.2. Initial Registry Contents

- o Parameter Name: "keys"
- o Parameter Description: Array of JWK values
- o Change Controller: IESG
- o Specification Document(s): Section 5.1 of [[this document]]

8.5. Media Type Registration**8.5.1. Registry Contents**

This specification registers the "application/jwk+json" and "application/jwk-set+json" Media Types [RFC2046] in the MIME Media Types registry [IANA.MediaType] in the manner described in RFC 6838 [RFC6838], 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: OpenID Connect, Salesforce, Google, Android, Windows Azure, W3C WebCrypto API, numerous others
- o Fragment identifier considerations: n/a
- 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 Provisional registration? No

- 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: OpenID Connect, Salesforce, Google, Android, Windows Azure, W3C WebCrypto API, numerous others
- o Fragment identifier considerations: n/a
- 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 Provisional registration? No

9. Security Considerations

All of the security issues that are pertinent to any cryptographic application must be addressed by JWS/JWE/JWK agents. Among these issues are protecting the user's asymmetric private and symmetric secret keys and employing countermeasures to various attacks.

9.1. Key Provenance and Trust

One should place no more trust in the data cryptographically secured by a key than in the method by which it was obtained and in the trustworthiness of the entity asserting an association with the key. Any data associated with a key that is obtained in an untrusted manner should be treated with skepticism. See Section 10.3 of [JWS] for security considerations on key origin authentication.

In almost all cases, applications make decisions about whether to

trust a key based on attributes bound to the key, such as names, roles, and the key origin, rather than based on the key itself. When an application is deciding whether to trust a key, there are several ways that it can bind attributes to a JWK. Two example mechanisms are PKIX [RFC5280] and JSON Web Token (JWT) [JWT].

For instance, the creator of a JWK can include a PKIX certificate in the JWK's "x5c" member. If the application validates the certificate and verifies that the JWK corresponds to the subject public key in the certificate, then the JWK can be associated with the attributes in the certificate, such as the subject name, subject alternative names, extended key usages, and its signature chain.

Also for instance, a JWT can be used to associate attributes with a JWK by referencing the JWK as a claim in the JWT. The JWK can be included directly as a claim value or the JWT can include a TLS-secured URI from which to retrieve the JWK value. Either way, an application that gets a JWK via a JWT claim can associate it with the JWT's cryptographic properties and use these and possibly additional claims in deciding whether to trust the key.

The security considerations in Section 12.3 of XML DSIG 2.0 [W3C.NOTE-xmlsig-core2-20130411] about the strength of a digital signature depending upon all the links in the security chain also apply to this specification.

The TLS Requirements in Section 8 of [JWS] also apply to this specification, except that the "x5u" JWK member is the only feature defined by this specification using TLS.

9.2. Preventing Disclosure of Non-Public Key Information

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. Of course, this requires that there be a secure way to obtain the key used to encrypt the non-public key information to the intended party and a secure way for that party to obtain the corresponding decryption key.

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

9.3. RSA Private Key Representations and Blinding

The RSA Key blinding operation [Kocher], which is a defense against some timing attacks, requires all of the RSA key values "n", "e", and

"d". However, some RSA private key representations do not include the public exponent "e", but only include the modulus "n" and the private exponent "d". This is true, for instance, of the Java RSAPrivateKeySpec API, which does not include the public exponent "e" as a parameter. So as to enable RSA key blinding, such representations should be avoided. For Java, the RSAPrivateCrtKeySpec API can be used instead. Section 8.2.2(i) of the Handbook of Applied Cryptography [HAC] discusses how to compute the remaining RSA private key parameters, if needed, using only "n", "e", and "d".

9.4. Key Entropy and Random Values

See Section 10.1 of [JWS] for security considerations on key entropy and random values.

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), January 2015.

[JWE]

Jones, M. and J. Hildebrand, "JSON Web Encryption (JWE)", draft-ietf-jose-json-web-encryption (work in progress), January 2015.

[JWS]

Jones, M., Bradley, J., and N. Sakimura, "JSON Web Signature (JWS)", draft-ietf-jose-json-web-signature (work in progress), January 2015.

- [RFC20] Cerf, V., "ASCII format for Network Interchange", RFC 20, October 1969.
- [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.
- [RFC4648] Josefsson, S., "The Base16, Base32, and Base64 Data Encodings", RFC 4648, October 2006.
- [RFC4945] Korver, B., "The Internet IP Security PKI Profile of IKEv1/ISAKMP, IKEv2, and PKIX", RFC 4945, August 2007.
- [RFC4949] Shirey, R., "Internet Security Glossary, Version 2", RFC 4949, August 2007.
- [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.
- [RFC7159] Bray, T., "The JavaScript Object Notation (JSON) Data Interchange Format", RFC 7159, March 2014.
- [UNICODE] The Unicode Consortium, "The Unicode Standard", 1991-, <<http://www.unicode.org/versions/latest/>>.

10.2. Informative References

- [DSS] National Institute of Standards and Technology, "Digital Signature Standard (DSS)", FIPS PUB 186-4, July 2013.
- [HAC] Menezes, A., van Oorschot, P., and S. Vanstone, "Handbook of Applied Cryptography", CRC Press, 1996, <<http://cacr.uwaterloo.ca/hac/about/chap8.pdf>>.
- [JWT] Jones, M., Bradley, J., and N. Sakimura, "JSON Web Token (JWT)", draft-ietf-oauth-json-web-token (work in progress), January 2015.
- [Kocher] Kocher, P., "Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems", In Proceedings of the 16th Annual International Cryptology Conference Advances in Cryptology, Springer-Verlag, pp. 104-113, 1996.
- [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.
- [RFC5226] Narten, T. and H. Alvestrand, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 5226, May 2008.
- [RFC6030] Hoyer, P., Pei, M., and S. Machani, "Portable Symmetric Key Container (PSKC)", RFC 6030, October 2010.
- [RFC6838] Freed, N., Klensin, J., and T. Hansen, "Media Type Specifications and Registration Procedures", BCP 13, RFC 6838, January 2013.
- [W3C.CR-WebCryptoAPI-20141211] Sleevi, R. and M. Watson, "Web Cryptography API", World Wide Web Consortium Candidate Recommendation CR-WebCryptoAPI-20141211, December 2014, <<http://www.w3.org/TR/2014/CR-WebCryptoAPI-20141211/>>.
- [W3C.NOTE-xmlsig-core2-20130411] Eastlake, D., Reagle, J., Solo, D., Hirsch, F., Roessler, T., Yiu, K., Datta, P., and S. Cantor, "XML Signature Syntax and Processing Version 2.0", World Wide Web

Consortium Note NOTE-xmlsig-core2-20130411, April 2013,
<<http://www.w3.org/TR/2013/NOTE-xmlsig-core2-20130411/>>.

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. (Line breaks within values are 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  
QvzqY368QQMicAtaSqzs8KJZgnYb9c7d0zgdAZHzu6QMqvRL5hajrnl91CbOpbI  
SD08qNLyrdkt-bFTWhAI4vMQFh6WeZu0fM4lFd2NcRwr3XPksINHaQ-G_xBniIqb  
w0LsljF44-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 within values 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-Bk0KWNGDjJHZDdDmFhW3AN7lI-puxk_mHZGJ1lrx  
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 within values are for display purposes only.)

```
{ "keys":  
  [  
    { "kty": "oct",  
      "alg": "A128KW",  
      "k": "GawggguFyGrWKav7AX4VKUg" },  
  
    { "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 an RSA public key and as an X.509 certificate using the "x5c" parameter (with line breaks within values for display purposes only):

```
{ "kty": "RSA",
  "use": "sig",
  "kid": "1b94c",
  "n": "vrjOfz9Ccdgx5nQudyhdoR17V-IubWMeOZCwX_jj0hgAsz2J_pqYW08
  PLbK_PdiVGKPrqzmDI7sA25VENHULuCLNwBuUiCO11_-7dYbsr4iJmG0Q
  u2j8DsVyTlazzJC_NG84Ty5KKthuCaPod7iI7w0LK9orSMhBEwwZDCxTWq4a
  YWachc8t-emd9qOvWtVMDC2BXksRngh6X5bUYLy6AyHKvj-nUylwgzjYQDwH
  MTplCoLtU-o-8SNnZltnRoGE9uJkBLdh5gFENabWnU5mlZqZPdwS-qo-meMv
  VfJb6jJVWRpl2SutCnYG2C32qvbWbjZ_jBPD5eunqsIoIvQ",
  "e": "AQAB",
  "x5c": [
    "MIIDQjCCAiqgAwIBAgIGATz/FuLiMa0GCSqGSIB3DQEBBQUAMGIXCzAJB
    gNVBAYTAlVTMQswCQYDVQQIEwJDTzEPMA0GA1UEBxMGRGVudmVyMRwwGgYD
    VQQKEsNQaW5nIElkZW50aXR5IENvcnAuMRcwFQYDVQQDEw5CcmlhbiBDYW1
    wYmVsbDAeFw0xMzAyMjE5MTVaFw0xODA4MTQyMjE5MTVaMGIXCzAJBg
    NVBAYTAlVTMQswCQYDVQQIEwJDTzEPMA0GA1UEBxMGRGVudmVyMRwwGgYD
    VQQKEsNQaW5nIElkZW50aXR5IENvcnAuMRcwFQYDVQQDEw5CcmlhbiBDYW1
    wYmVsbDCCASIwDQYJKoZIhvcNAQEBBQADggEPADCCAQoCggEBAL64zn8/QnH
    YMeZ0LncOXAEdelfiLmljHjmQsF/449IYALM9if6amFtPDy2yvv3YlRij66
    s5gyLCyO7ANuVRJxlNbgizcAblIgjtdf/u3WG7K+IiZhtELto/A7Fck9Ws6
    SQqvRvOE8uSirYbgmj6He4iO8NCyvaK0jIQRMGQwsU1quGmFgHIXPLfnpn
    fajrlrVTawtgV5LEZ4Iel+W1GC8ugMhyr4/p1MtcIM42EA8BzE6ZQqC7VPq
    PveJz2dbZkaBhPbiZAS3YeYBRDWmlp1OztWamT3cEvqqPpnjL1XyW+oyVVk
    aZdklLQp2Btgt9qr2lm42f4wTw+Xrp6rCKNb0CAWEAATANBgkqhkiG9w0BA
    QUFAAOCAQEAh8zGlfsIcI0o3rYDPBB07aXNswb4ECNIG0CETTUXmXl9KUL
    +9gGlqCz5iWLOgWsnrcKcY0vXPG9Jlr9AqBNTqNgHq2G03X09266X5CpOe1
    zFo+Owblzxt3PehFdfQJ610CDLEaS9V9Rqp17hCyybEpOGVwe8fnk+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 authenticated encryption operation, formatted as a JWK (with line breaks within values for display purposes only):

```
{
  "kty": "RSA",
  "kid": "juliet@capulet.lit",
  "use": "enc",
  "n": "t6Q8PWSildkJj9hTP8hNYFlvadM7DflW9mWepOJhJ66w7nyoK1gPNqFMSQRy
    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-dVJ7jahliI7LyckrpTmrM8dWBo4_PMaenNnPiqgO0xnu
    ToxutRZJfJvG4Ox4ka3GORQd9CsCZ2vsUDmsXOfUENoyMqADC6p1M3h33tsu
    rY15k9qMSPG9OX_IJAXmxzAh_tWiZOWk2K4yxH9tS3Lq1yX8ClEWmeRDkK2a
    hecG85-oLKQt5VEpWHKmjiOi_gJSdSgqcn96X52esAQ",
  "p": "2rnSOV4hKSN8sS4CgcQHFbs08XboFDqKum3sc4h3GRxrTmqdl1ZK9uw-PIHf
    QP0FkxXVrx-WE-ZEbrqivH_2iCLUS7wAl6XvART1KkIaUxPPSYB9yk31s0Q8
    UK96E3_OrADAYtAJs-M3JxCLfNgqh56HDnETTQhH3rCT5T3yJws",
  "q": "lu_RiFDP7LBYh3N4GXLT9OpSKYP0uQZyiaZwBtOCBNJgQxaj10RWjsZu0c6I
    edis4S7B_coSKB0Kj9PaPaBzg-IySRvvcQuPamQu66riMhjVtG6TlV8CLCYK
    rY152ziqK0E_ym2QnkwsUX7eYTB7LbAHRK9GqocDE5B0f808I4s",
  "dp": "KkMTWqBUefVwZ2_DbjlPQqyHSHjj90L5x_MOzqYAJMcLMZtbUtwKqvVDq3
    tbEo3ZiCohbDtt6SbfmWzggabpQxNxuBpo0Of_a_HgMXK_lhqigI4y_kqS1w
    Y52IwjUn5rgRrJ-yYolh4lKR-vz2pYhEAEYrhtWtxVqLCRviD6c",
  "dq": "AvfS0-gRxvn0bwJoMSnFXYcK1WnuEjQFluMGfwGitQBWtfZ1Er7tlxDkbN9
    GQTb9yqpDoYaN06H7CFtrkxhJIBQaj6nkF5KKS3TQtQ5qCzkOkmxIe3KRbBy
    mXxkb5qwUpX5ELD5xFc6FeiafWYY63TmmeEAu_lRFCOJ3xDea-ots",
  "qi": "lSQi-w9CpyURemErPlRsBLk7wNtOvs5EQpPqmuMvqW57NBUCzScEoPwmUqq
    abu9V0-Py4dQ57_bapoKRulR90bvFnU63SHWEFglZQvJDMeAvmj4sm-Fp0o
    Yu_neotgQ0hzbI5gry7ajdYy9-2lNx_76aBZoOUu9HCJ-UsfSOI8"
}
```

The octets representing the Plaintext used in this example (using JSON array notation) 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. JOSE 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 Input ("p2s") value is [217, 96, 147, 112, 150, 117, 70, 247, 127, 8, 155, 137, 174, 42, 80, 215],
- o the Iteration Count ("p2c") value is 4096,
- o authenticated encryption is performed on the Plaintext using the AES_128_CBC_HMAC_SHA_256 algorithm to produce the Ciphertext and the Authentication Tag, 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 (with line breaks for display purposes only):

```
eyJhbGciOiJQQkVtMlIUZiI1NitBMTI4SlciLCJwMnMiOiIyV0NUY0paMVJ2ZF9DSn
VKcmlwUTF3IiwicDJjIjo0MDk2LCJlbnMiOiJBMTI4Q0JDLUhTMjU2IiwiaWF0Ijoi
andrK2pzb24ifQ
```

C.3. Content Encryption Key (CEK)

Generate a 256 bit random Content Encryption Key (CEK). In this example, the value (using JSON array notation) 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 Derivation

Derive a key from a shared passphrase using the PBKDF2 algorithm with HMAC SHA-256 and the specified Salt and Iteration Count values and a 128 bit requested output key size to produce the PBKDF2 Derived 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 Salt value (UTF8(Alg) || 0x00 || Salt Input) is:

[80, 66, 69, 83, 50, 45, 72, 83, 50, 53, 54, 43, 65, 49, 50, 56, 75, 87, 0, 217, 96, 147, 112, 150, 117, 70, 247, 127, 8, 155, 137, 174, 42, 80, 215].

The resulting PBKDF2 Derived Key value is:

[110, 171, 169, 92, 129, 92, 109, 117, 233, 242, 116, 233, 170, 14, 24, 75]

C.5. Key Encryption

Encrypt the CEK with the "A128KW" algorithm using the PBKDF2 Derived Key. The resulting JWE Encrypted Key value is:

[78, 186, 151, 59, 11, 141, 81, 240, 213, 245, 83, 211, 53, 188, 134, 188, 66, 125, 36, 200, 222, 124, 5, 103, 249, 52, 117, 184, 140, 81, 246, 158, 161, 177, 20, 33, 245, 57, 59, 4]

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

TrqXOWuNUfDV9VPTNbyGvEJ9JMjefAVn-TRluIxR9p6hsRQh9Tk7BA

C.6. 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.7. 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.8. Content Encryption

Perform authenticated encryption on the Plaintext with the AES_128_CBC_HMAC_SHA_256 algorithm 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:

[208, 113, 102, 132, 236, 236, 67, 223, 39, 53, 98, 99, 32, 121, 17, 236]

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

```
AwhB8lxlRkKjFn02LGWEqg27H4Tg9fyZAbFv3p5ZicHpj64QyHC44qqLz3JEmnZTgQo
wIqZJ13jbyHB8LgePiQJ1hf6M2HPLgzW8L-mEeQ0jvDUTrE07NtOerBk8bwBQyZ6g
0kQ3DEOIglfYxV8-FJvNBYwbqN1Bck6d_i7OtjSHV-8Dirp-3JcRIe05YKy3Oi34Z_
GOiAc1EK2lB1lc_AE1lPII_wvvtRiUiG8YofQXakWd1_098Kap-UgmyWPfreUJ3lJP
nbD4Ve95owEfMGLOPflo2MnjaTDCwQokoJ_xplQ2vNPz8iguLCHBoKllyQFJL2mOWB
wqhBo9Oj-0800as5mmLsvQMTf1IrIEbbTMzHMBZ8EFW9fWwwFu0DWQJGkMNMhMBZQ-3
lvqTc-M6-gWA6D8PDhONfP2Oib2HGizwG1iEaX8GRyUpfLuljCLiElDkGOewhKuKkZ
h04DKNM5Nbugf2atmU9OP0Ldx5peCUtRG1gMVl7Qup5ZXHTjgPDr5b2N731UooCGAU
qHdgGhg0JVJ_ObCTdjsH4CF1SJsdUhrXvYx3HJh2Xd7CwJRzU_3Y1GxYU6-s3GFPbi
rfqqEipJDBTHpcoCmyrWYjYHFgnlqBZRotRrS95g8F95bRXqsaDY7UgQGwBQBwy665
d0zpvTasvfXf_c0MwAl-neFaKOW_Px6g4EUDjG1GWSXV9cLStLw_0ovdApDIFLHYHe
PyagyHjouQUuGiq7BsYwYrwaF06tgB8hV8omLNfMEMDPJaZUzMuHw6tBDWgkzD-ts_
ub9hxrPJ4UsOWnt5rGUyoN2N_c1-TQlXxm5oto14MxnoAyBQBpwIEgSH3Y4ZhwKBhH
PjSo0cdwuNdYbGPpb-YUvF-2NZzODiQ10vWQBRSbPWYz_xbGkgD504LrtqRwCO7CC
_CyyURilSessPVsMJRX_U4LFEoc82TiDdqjKOjRUfKK5rqLi8nBE9soQ0DSaOoFQZi
GrBrqxDSNYiAYAmxxkos-i3nX4qtByVx85sCE5U_0MqG7COxZWMOPeFrDaepUV-cOy
rvoUIng8i8ljkBKxEtY2BgPegKBYCxsAUcAkKamSCC9AiBxA0UOHytTqtlvMks07AE
hNC2-YzPyxlFkhMoS4LLe6E_pFsMlmjA6PlNSge9C5G5tETyXGAN6blxZbHtmwrPSc
ro9LWhVmAA7_bxYObnFUxgWtK4vzzQBjZJ36UTk4OTB-JvKWgfvWCFsaw5WCHj6Oo
4jp07d2yN7WMfAj2hTEabz9wumQ0TMhBduZ-QON3pYObSy7TSC1vVme0NJrwF_cJRe
hKTFmdlXGVldPxZCplr7ZQqRQhF8JP-14mEQVnCaWGn9ONHlemczGOS-A-wwtnmwjI
BlV_vgJrf4FdpV-4hUk4-QLpu3-1lWFxrtZKcggq3tWTduRo5_QebQbUUT_VSCgsFc
OmyWkoj56lbxthN19hq1XGWbLGfrrR6MWh23vk01zn8FVwi7uFwEnRYSafsnWLa1Z5
TpBj9GvAdl2H9NHwzpB5NqHpZNkQ3NMDj13Fn8fz00JB83Etbm_tnFQfcb13X3bJ15
Cz-Ww1MGhvIpGGnMBT_AdP9xSiYAM9dQ1yeVXk-AIGWBULN5uyWSGyCxp0cJwx7HxM
38z0UIeBu-MytL-eqndM7LxytsVzCbJOTSVRmhYEMiZUAnSlgs7uMQAGRdgrIElTJE
SGMjb_4bZq9s6VelLKkSi0_QDsraBaLe55UY0zf4ZSfOV5PMYPtOcW_dcNPlxLgNA
D1BFX_Z9kAdMZQW6fAmsfFle0zAoMe4l9pMESH0JB4sJGdCKtQXj1cXNydyYozF718
H00BV_Er7zd6VtIw0MxwkFCTatsv_R-GsBCH218RgVPsfYhwVuT8R4HarpzsDBufC4
r8_c8fc9Z278sQ081jFjOja6L2x0N_ImzFNXU6xwO-Ska-QeuvYZ3X_L31ZOX4Llp-
7QSfgDoHnOxFv1Xws-D5mDHD3zxOup2b2TppdKTZb9eW2vxUVviM8OI9atBfPKMGAO
v9omA-6vv5IxUH0-lWmiHLQ_g8vnswp-Jav0c4t6URVUzujNOoNd_CBGgVnHiJTCHL
88LQxssqLHHIu4Fz-U2SGnlxGTj0-ihit2ELGRv4v08E1BosTmf0cx3qgG0Pq0eOLBD
IHsrdZ_CCAiTc0HVkMbyqlM6qEhM-q5P6ylQCirwg
```

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

```
0HFmhOzsQ98nNWJjIHkR7A
```

C.9. Complete Representation

Assemble the final representation: The JWE Compact Serialization of this result, as defined in Section 7.1 of [JWE], 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:

AwhB8lxlRkJfFn02LGEWegg27H4Tg9fyZAbFv3p5ZicHpj64QyHC44qqlZ3JEmnZTgQo
 wIqZJ13jbyHB8LgePiQUJ1hf6M2HPLgzW8L-mEEQ0jvDUTRe07NtOerBk8bwBQyZ6
 0kQ3DEOIglfYxV8-FJvNBYwbqN1Bck6d_i70tjSHV-8DIrp-3JcRIe05YKy3Oi34Z
 GOiAc1EK21B1lc_AE11PII_wvvtrRiUiG8YofQXakWd1_098Kap-UgmyWPfreUJ3lJP
 nbd4Ve95owEfMGLoSF20MnjtaTDCWQokoJ_xplQ2vNPZ8iguLCHBoKllyQFJL2mOWB
 wqhBo9cJ-0800as5mmLsvQMtf1lrIEbbTMZHBMBZ8EFW9fWwwFu0DWQJGkMNNhMBZQ-3
 lvgTc-M6-gWA6D8PDhOnfP20ib2HGizWg1iEaX8GRYUpfLuljCLIEldKGOewhKuKkZ
 h04DKNM5Nbugf2atmU90P0Ldx5peCUTRGlgMv17Qup5ZXHTjgPDr5b2N731UooCGAU
 qHdgGhg0JVJ_ObCTdjsH4CF1SJsduhrXvYx3HJh2Xd7CwJRzU_3Y1GxYU6-s3GFPbi
 rfqqEipJDBTHpcoCmyrwyJYHFgnlqBZRotRrS95g8F95bRXqsaDY7UgQGwBQBwy665
 d0zpvTasvfXf_c0MWA1-neFaKOW_Px6g4EUDjG1GWSXV9cLStLw_0ovdApDIFLHYHe
 PyagyHjouQUuGiQ7BsYwYrwaF06tgB8hV8omLNFmEMDPJaZUZuMuHw6tBDwGkzD-tS_
 ub9hxrPj4UsoWnt5rGUyON2N_c1-TQlXxm5oto14MxnoAyBQBpwiEGSH3Y4ZhwKBhH
 PjSo0cdwuNdYbGPpb-YUVf-2NZzODiQ10vWQBRHSbPWYz_xbGkgD504LRtqRwCO7CC
 _CyyURilsEssPVsMJRX_U4LFE0c82TiDdqjK0jRUfKK5rqLi8nBE9soQ0DSaOoFQZi
 GrBrqxDsNYiAYAmxxkos-i3nX4qtByVx85sCE5U_0MqG7COxZWMOPEFrDaepUV-cOy
 rvoUIng8i8ljKBKxETY2BgPegKBYCxsAUcAkKamSCC9AiBxA0UOHYhTqt1vMksO7AE
 hNC2-YzPyx1FkhMoS4LLe6E_pFsMlmjA6P1NSge9C5G5tETYXGAn6blxZbHtmwrPSc
 ro9LWhVmAA7_bxYObnFUxgWtK4vz3QBJZ36UTk40TB-JvKXgfVWCFsaw5WCHj6Oo
 4jpf07d2yN7WMAfj2bTEabz9wumQ0TMhBduZ-QON3pYObSy7TSC1vVme0NjrwF_cJRe
 hKTFmdlXGVldPxZCplr7ZQqRQhF8JP-14mEQVnCaWgn9ONHlemczGOS-A-wwtnmwjI
 BlV_vgJrf4FdpV-4hUk4-QLpu3-1lWFxrtZKcgq3tWTduRo5_QebQbUUT_VSCgsFc
 OmyWkoj56lxbxthN19hq1XGwbLGfrrR6MWh23vk01zn8FVwi7uFwEnRYSafsnWLa1Z5
 TpBj9GvAdl2H9NHwzPb5NqHpZNkQ3NMDj13Fn8fz00JB83Etbm_tnFQfcbl3X3bJ15
 Cz-Ww1MGhvIpGGnMBT_AdP9xSIyAM9dQ1yeVXk-AIGWBULN5uyWSGyCxp0cJwx7HxM
 38z0UIeBu-MytL-eqndM7LxytsVzCbJOTSVRmhYEMIzUANs1gs7uMQAGRdgrIElTJE
 SGMjb_4bzq9s6Ve1LKkSi0_QDsRABaLe55UY0zF4ZSfOV5PMYptocwV_dcNPlxLgNA
 D1BFX_Z9kAdmZQW6fAmsfFle0zAoMe4l9pMESH0JB4sJGdCKtQXj1cXNyDDYozF718
 H00BV_Er7zd6VtIw0MxwkFCTatsv_R-GsBCH218RgVPsfYhwVuT8R4HarpzsDBufC4
 r8_c8fc9Z278sQ081jFj0ja6L2x0N_ImzFNXu6xwO-Ska-QuevYZ3X_L31ZOx4Llp-
 7QsfgDoHnOxFlXws-D5mDHD3zxOup2b2TppdKTZb9eW2vxUVviM8OI9atBfPKMGAO
 v9omA-6vv5IxUH0-1WmiHLQ_g8vnswp-Jav0c4t6URVUzujNoONd_CBGgVnHiJTCH1
 88LQXsqLHHI4Fz-U2SgnlxGTj0-ihit2ELGRv4vO8E1BosTmf0cx3qgG0Pq0eOLBD
 IhsrdZ_CCAiTC0HVkMbyq1M6qEhM-q5P6y1QCirwg.
 0HFmhOzsQ98nNWJjIHkR7A

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].

Thanks to Matt Miller for creating the encrypted key example and to Edmund Jay and Brian Campbell for validating the example.

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, Stephen Farrell, Joe Hildebrand, Edmund Jay, Stephen Kent, Ben Laurie, James Manger, Matt Miller, Kathleen Moriarty, Chuck Mortimore, Tony Nadalin, Axel Nennker, John Panzer, Eric Rescorla, Pete Resnick, Nat Sakimura, Jim Schaad, Ryan Sleevi, Paul Tarjan, Hannes Tschofenig, and Sean Turner.

Jim Schaad and Karen O'Donoghue chaired the JOSE working group and Sean Turner, Stephen Farrell, and Kathleen Moriarty 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]]

-41

- o Added Security Considerations text about binding attributes to keys.
- o Incorporated additional terms defined in the JWE spec by reference.

-40

- o Clarified the definitions of UTF8(String) and ASCII(String).
- o Stated that line breaks are for display purposes only in places where this disclaimer was needed and missing.
- o Updated the WebCrypto reference to refer to the W3C Candidate Recommendation.

-39

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

-38

- o Replaced uses of the phrase "JWK object" with "JWK".

-37

- o Updated the TLS requirements language to only require implementations to support TLS when they support features using TLS.
- o Restricted algorithm names to using only ASCII characters.
- o Updated the example IANA registration request subject line.

-36

- o Stated that if both "use" and "key_ops" are used, the information they convey MUST be consistent.
- o Clarified where white space and line breaks may occur in JSON objects by referencing Section 2 of RFC 7159.
- o Specified that registration reviews occur on the jose-reg-review@ietf.org mailing list.

-35

- o Used real values for examples in the IANA Registration Templates.

-34

- o Addressed IESG review comments by Pete Resnick, Stephen Farrell, and Richard Barnes.
- o Referenced RFC 4945 for PEM certificate delimiter syntax.

-33

- o Addressed secdir review comments by Stephen Kent for which resolutions had mistakenly been omitted in the previous draft.
- o Acknowledged additional contributors.

-32

- o Addressed Gen-ART review comments by Russ Housley.
- o Addressed secdir review comments by Stephen Kent.

-31

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

-30

- o Added references and cleaned up the reference syntax in a few places.
- o Applied minor wording changes to the Security Considerations section.

-29

- o Replaced the terms JWS Header, JWE Header, and JWT Header with a single JOSE Header term defined in the JWS specification. This also enabled a single Header Parameter definition to be used and reduced other areas of duplication between specifications.

-28

- o Revised the introduction to the Security Considerations section.
- o Refined the text about when applications using encrypted JWKs and JWK Sets would not need to use the "cty" header parameter.

-27

- o Added an example JWK early in the draft.
- o Described additional security considerations.
- o Added the "x5t#S256" (X.509 Certificate SHA-256 Thumbprint) JWK member.
- o Addressed a few editorial issues.

-26

- o Referenced Section 6 of RFC 6125 for TLS server certificate identity validation.
- o Deleted misleading non-normative phrase from the "use" description.

- o Noted that octet sequences are depicted using JSON array notation.
- o Updated references, including to W3C specifications.

-25

- o Updated WebCrypto reference to refer to W3C Last Call draft.

-24

- o Corrected the authentication tag value in the encrypted key example.
- o Updated the JSON reference to RFC 7159.

-23

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

-22

- o Corrected RFC 2119 terminology usage.
- o Replaced references to draft-ietf-json-rfc4627bis with RFC 7158.

-21

- o Replaced the "key_ops" values "wrap" and "unwrap" with "wrapKey" and "unwrapKey" to match the "KeyUsage" values defined in the current Web Cryptography API editor's draft.
- o Compute the PBES2 salt parameter as (UTF8(Alg) || 0x00 || Salt Input), where the "p2s" Header Parameter encodes the Salt Input value and Alg is the "alg" Header Parameter value.
- o Changed some references from being normative to informative, addressing issue #90.

-20

- o Renamed "use_details" to "key_ops" (key operations).
- o Clarified that "use" is meant for public key use cases, "key_ops" is meant for use cases in which public, private, or symmetric keys may be present, and that "use" and "key_ops" should not be used together.

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

-19

- o Added optional "use_details" (key use details) JWK member.
- o Reordered the key selection parameters.

-18

- o Changes to address editorial and minor issues #68, #69, #73, #74, #76, #77, #78, #79, #82, #85, #89, and #135.
- o Added and used Description registry fields.

-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 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: July 20, 2015

M. Jones
Microsoft
J. Bradley
Ping Identity
N. Sakimura
NRI
January 16, 2015

JSON Web Signature (JWS)
draft-ietf-jose-json-web-signature-41

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 July 20, 2015.

Copyright Notice

Copyright (c) 2015 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	6
3. JSON Web Signature (JWS) Overview	7
3.1. JWS Compact Serialization Overview	8
3.2. JWS JSON Serialization Overview	8
3.3. Example JWS	9
4. JOSE Header	10
4.1. Registered Header Parameter Names	11
4.1.1. "alg" (Algorithm) Header Parameter	11
4.1.2. "jku" (JWK Set URL) Header Parameter	11
4.1.3. "jwk" (JSON Web Key) Header Parameter	11
4.1.4. "kid" (Key ID) Header Parameter	12
4.1.5. "x5u" (X.509 URL) Header Parameter	12
4.1.6. "x5c" (X.509 Certificate Chain) Header Parameter	12
4.1.7. "x5t" (X.509 Certificate SHA-1 Thumbprint) Header Parameter	13
4.1.8. "x5t#S256" (X.509 Certificate SHA-256 Thumbprint) Header Parameter	13
4.1.9. "typ" (Type) Header Parameter	13
4.1.10. "cty" (Content Type) Header Parameter	14
4.1.11. "crit" (Critical) Header Parameter	14
4.2. Public Header Parameter Names	15
4.3. Private Header Parameter Names	15
5. Producing and Consuming JWSs	15
5.1. Message Signature or MAC Computation	15
5.2. Message Signature or MAC Validation	16
5.3. String Comparison Rules	18
6. Key Identification	19
7. Serializations	19
7.1. JWS Compact Serialization	20
7.2. JWS JSON Serialization	20
7.2.1. General JWS JSON Serialization Syntax	20
7.2.2. Flattened JWS JSON Serialization Syntax	22
8. TLS Requirements	23
9. IANA Considerations	23
9.1. JSON Web Signature and Encryption Header Parameters Registry	24

9.1.1.	Registration Template	25
9.1.2.	Initial Registry Contents	25
9.2.	Media Type Registration	27
9.2.1.	Registry Contents	27
10.	Security Considerations	28
10.1.	Key Entropy and Random Values	28
10.2.	Key Protection	29
10.3.	Key Origin Authentication	29
10.4.	Cryptographic Agility	29
10.5.	Differences between Digital Signatures and MACs	29
10.6.	Algorithm Validation	30
10.7.	Algorithm Protection	30
10.8.	Chosen Plaintext Attacks	31
10.9.	Timing Attacks	31
10.10.	Replay Protection	31
10.11.	SHA-1 Certificate Thumbprints	31
10.12.	JSON Security Considerations	32
10.13.	Unicode Comparison Security Considerations	32
11.	References	33
11.1.	Normative References	33
11.2.	Informative References	34
Appendix A.	JWS Examples	36
A.1.	Example JWS using HMAC SHA-256	36
A.1.1.	Encoding	36
A.1.2.	Validating	38
A.2.	Example JWS using RSASSA-PKCS-v1_5 SHA-256	39
A.2.1.	Encoding	39
A.2.2.	Validating	41
A.3.	Example JWS using ECDSA P-256 SHA-256	42
A.3.1.	Encoding	42
A.3.2.	Validating	44
A.4.	Example JWS using ECDSA P-521 SHA-512	44
A.4.1.	Encoding	44
A.4.2.	Validating	46
A.5.	Example Unsecured JWS	46
A.6.	Example JWS using General JWS JSON Serialization	47
A.6.1.	JWS Per-Signature Protected Headers	48
A.6.2.	JWS Per-Signature Unprotected Headers	48
A.6.3.	Complete JOSE Header Values	48
A.6.4.	Complete JWS JSON Serialization Representation	49
A.7.	Example JWS using Flattened JWS JSON Serialization	49
Appendix B.	"x5c" (X.509 Certificate Chain) Example	50
Appendix C.	Notes on implementing base64url encoding without padding	52
Appendix D.	Notes on Key Selection	53
Appendix E.	Negative Test Case for "crit" Header Parameter	54
Appendix F.	Detached Content	55
Appendix G.	Acknowledgements	55

Appendix H. Document History	56
Authors' Addresses	67

1. Introduction

JSON Web Signature (JWS) represents content secured with digital signatures or Message Authentication Codes (MACs) using JavaScript Object Notation (JSON) [RFC7159] based data structures. The JWS cryptographic mechanisms provide integrity protection for an arbitrary sequence of octets. See Section 10.5 for a discussion on the differences between Digital Signatures and MACs.

Two closely related serializations for JWSS 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 JWSS 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", "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, where String is a sequence of zero or more Unicode [UNICODE] characters.

ASCII(String) denotes the octets of the ASCII [RFC20] representation of String, where String is a sequence of zero or more ASCII characters.

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

2. Terminology

These terms are defined by this specification:

JSON Web Signature (JWS)

A data structure representing a digitally signed or MACed message.

JOSE Header

JSON object containing the parameters describing the cryptographic operations and parameters employed. The JOSE Header is comprised of a set of 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

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

Header Parameter

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

JWS Protected Header

JSON object that contains the Header Parameters that are integrity protected by the JWS Signature digital signature or MAC operation. For the JWS Compact Serialization, this comprises the entire JOSE Header. For the JWS JSON Serialization, this is one component of the JOSE Header.

JWS Unprotected Header

JSON object that contains the Header Parameters that are not integrity protected. This can only be present when using the JWS JSON Serialization.

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) and without the inclusion of any line breaks, white space, or other additional characters. Note that the base64url encoding of the empty octet sequence is the empty string. (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 a compact, URL-safe string.

JWS JSON Serialization

A representation of the JWS as a JSON object. 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 optimized for compactness nor URL-safe.

Unsecured JWS

A JWS that provides no integrity protection. Unsecured JWSs use the "alg" value "none".

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.

These terms defined by the JSON Web Encryption (JWE) [JWE] specification are incorporated into this specification: "JSON Web Encryption (JWE)", "JWE Compact Serialization", and "JWE JSON Serialization".

These terms defined by the Internet Security Glossary, Version 2 [RFC4949] are incorporated into this specification: "Digital Signature" and "Message Authentication Code (MAC)".

3. JSON Web Signature (JWS) Overview

JWS represents digitally signed or MACed content using JSON data structures and base64url encoding. These JSON data structures MAY contain white space and/or line breaks before or after any JSON values or structural characters, in accordance with Section 2 of RFC

7159 [RFC7159]. A JWS represents these logical values (each of which is defined in Section 2):

- o JOSE Header
- o JWS Payload
- o JWS Signature

For a JWS, the JOSE Header members are the union of the members of these values (each of which is defined in Section 2):

- o JWS Protected Header
- o JWS Unprotected Header

This document defines two serializations for JWSs: 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, since JSON lacks a way to directly represent arbitrary octet sequences.

3.1. JWS Compact Serialization Overview

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

In the JWS Compact Serialization, a JWS is represented as the concatenation:

```
BASE64URL(UTF8(JWS Protected Header)) || '.' ||  
BASE64URL(JWS Payload) || '.' ||  
BASE64URL(JWS Signature)
```

See Section 7.1 for more information about the JWS Compact Serialization.

3.2. JWS JSON Serialization Overview

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 JOSE Header are the union of the members of the JWS Protected Header and the JWS Unprotected Header values that are present.

In the JWS JSON Serialization, a JWS is represented as a JSON object containing some or all of these four members:

```
"protected", with the value BASE64URL(UTF8(JWS Protected Header))
"header", with the value JWS Unprotected Header
"payload", with the value BASE64URL(JWS Payload)
"signature", with the value BASE64URL(JWS Signature)
```

The three base64url encoded result strings and the JWS Unprotected Header value are 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.

3.3. 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 [RFC2104, SHS] 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):

```
eyJpc3MiOiJqb2UiLA0KICJleHAiOiJleMDA4MTkzODAsDQogImh0dHA6Ly9leGft
cGxlLnNvbS9pc19yb290Ijp0cnVlfQ
```

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_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
cGxlLnNvbS9pc19yb290Ijp0cnVlfQ
.
dBjftJeZ4CVP-mB92K27uhbUJU1plr_wWlgFWFOEjXk
```

See Appendix A for additional examples, including examples using the JWS JSON Serialization in Sections A.6 and A.7.

4. JOSE Header

For a JWS, the members of the JSON object(s) representing the JOSE 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 JOSE Header MUST be unique; JWS parsers 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].

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 4.1.11, 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 for use in JWSs are registered in the IANA JSON Web Signature and Encryption Header Parameters registry defined in Section 9.1, 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 JWS Signature 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 ASCII string containing a StringOrURI value. This Header Parameter MUST be present and 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 6 of RFC 6125 [RFC6125]. Also, see Section 8 on TLS requirements. 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. "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. The structure of the "kid" value is unspecified. Its value **MUST** be a case-sensitive string. Use of this Header Parameter is **OPTIONAL**.

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

4.1.5. "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, with each certificate delimited as specified in Section 6.1 of RFC 4945 [RFC4945]. 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 6 of RFC 6125 [RFC6125]. Also, see Section 8 on TLS requirements. Use of this Header Parameter is **OPTIONAL**.

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** validate the certificate chain according to RFC 5280 [RFC5280] and consider the certificate or certificate chain to be invalid if any validation failure occurs. Use of this Header Parameter is **OPTIONAL**.

See Appendix B for an example "x5c" value.

4.1.1.7. "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. Note that certificate thumbprints are also sometimes known as certificate fingerprints. Use of this Header Parameter is OPTIONAL.

4.1.1.8. "x5t#S256" (X.509 Certificate SHA-256 Thumbprint) Header Parameter

The "x5t#S256" (X.509 Certificate SHA-256 Thumbprint) Header Parameter is a base64url encoded SHA-256 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. Note that certificate thumbprints are also sometimes known as certificate fingerprints. Use of this Header Parameter is OPTIONAL.

4.1.1.9. "typ" (Type) Header Parameter

The "typ" (type) Header Parameter is used by JWS applications to declare the MIME Media Type [IANA.MediaType] of this complete JWS. This is intended for use by the application when more than one kind of object could be present in an application data structure that can contain a JWS; the application can use this value to disambiguate among the different kinds of objects that might be present. It will typically not be used by applications when the kind of object is already known. This parameter is ignored by JWS implementations; any processing of this parameter is performed by the JWS application. Use of this Header Parameter is OPTIONAL.

Per RFC 2045 [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 producers 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; whereas, the media type "application/example;part=1/2" cannot be shortened to "example;part=1/2".

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.

4.1.10. "cty" (Content Type) Header Parameter

The "cty" (content type) Header Parameter is used by JWS applications to declare the MIME Media Type [IANA.MediaType] of the secured content (the payload). This is intended for use by the application when more than one kind of object could be present in the JWS payload; the application can use this value to disambiguate among the different kinds of objects that might be present. It will typically not be used by applications when the kind of object is already known. This parameter is ignored by JWS implementations; any processing of this parameter is performed by the JWS application. Use of this Header Parameter is OPTIONAL.

Per RFC 2045 [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 producers 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; whereas, the media type "application/example;part=1/2" cannot be shortened to "example;part=1/2".

4.1.11. "crit" (Critical) Header Parameter

The "crit" (critical) Header Parameter indicates that extensions to the initial RFC versions of [[this specification]] and [JWA] are being used that MUST be understood and processed. Its value is an array listing the Header Parameter names present in the JOSE Header that use those extensions. If any of the listed extension Header Parameters are not understood and supported by the recipient, then the JWS is invalid. Producers MUST NOT include Header Parameter names defined by the initial RFC versions of [[this specification]] or [JWA] for use with JWS, duplicate names, or names that do not occur as Header Parameter names within the JOSE Header in the "crit" list. Producers MUST NOT use the empty list "[]" as the "crit" value. Recipients MAY consider the JWS to be invalid if the critical list contains any Header Parameter names defined by the initial RFC versions of [[this specification]] or [JWA] for use with JWS, or

any other constraints on its use are violated. When used, this Header Parameter MUST be integrity protected; therefore, it MUST occur only within the JWS Protected Header. 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 9.1 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 Signature or MAC Computation

To create a JWS, the following steps are performed. 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 the JSON object(s) containing the desired set of Header Parameters, which together comprise the JOSE Header: the JWS Protected Header and/or the JWS Unprotected Header.
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 JOSE 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. If the JWS JSON Serialization is being used, repeat this process (steps 3-6) for each digital signature or MAC operation being performed.
8. 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 are performed. 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.

When there are multiple JWS Signature values, it is an application decision which of the JWS Signature values must successfully validate for the JWS to be accepted. In some cases, all must successfully validate or the JWS will be considered invalid. In other cases, only a specific JWS Signature value needs to be successfully validated. However, in all cases, at least one JWS Signature value MUST successfully validate or the JWS MUST be considered invalid.

1. Parse the JWS representation to extract the serialized values for the components of the JWS. When using the JWS Compact Serialization, these components are the base64url encoded representations of the JWS Protected Header, the JWS Payload, and the JWS Signature, and when using the JWS JSON Serialization, these components also include 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, with each value being separated from the next by a single period ('.') character, resulting in exactly two delimiting period characters being used. The JWS JSON Serialization is described in Section 7.2.
2. Base64url decode the encoded representation of the JWS Protected Header, following the restriction that no line breaks, white space, or other additional characters have been used.
3. Verify that the resulting octet sequence is a UTF-8 encoded representation of a completely valid JSON object conforming to RFC 7159 [RFC7159]; let the JWS Protected Header be this JSON object.
4. If using the JWS Compact Serialization, let the JOSE Header be the JWS Protected Header. Otherwise, when using the JWS JSON Serialization, let the JOSE 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. During this step, verify that the resulting JOSE Header does 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 object values that together comprise the JOSE Header.
5. Verify that the implementation understands and can process all fields that it is required to support, whether required by this specification, by the algorithm being used, or by the "crit" Header Parameter value, and that the values of those parameters are also understood and supported.
6. Base64url decode the encoded representation of the JWS Payload, following the restriction that no line breaks, white space, or other additional characters have been used.
7. Base64url decode the encoded representation of the JWS Signature, following the restriction that no line breaks, white space, or other additional characters have been used.

8. Validate the JWS Signature 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. See Section 10.6 for security considerations on algorithm validation. Record whether the validation succeeded or not.
9. If the JWS JSON Serialization is being used, repeat this process (steps 4-8) for each digital signature or MAC value contained in the representation.
10. If none of the validations in step 9 succeeded, then the JWS MUST be considered invalid. Otherwise, in the JWS JSON Serialization case, return a result to the application indicating which of the validations succeeded and failed. In the JWS Compact Serialization case, the result can simply indicate whether or not the JWS was successfully validated.

Finally, note that it is an application decision which algorithms may be used in a given context. Even if a JWS can be successfully validated, unless the algorithm(s) used in the JWS are acceptable to the application, it SHOULD consider the JWS to be invalid.

5.3. String Comparison Rules

Processing a JWS inevitably requires comparing known strings to members and values in JSON objects. For example, in checking what the algorithm is, the Unicode string "alg" will be checked against the member names in the JOSE Header to see if there is a matching Header Parameter name. The same process is then used to determine if the value of the "alg" Header Parameter represents a supported algorithm.

The JSON rules for doing member name comparison are described in Section 8.3 of RFC 7159 [RFC7159]. Since the only string comparison operations that are performed are equality and inequality, the same rules can be used for comparing both member names and member values against known strings.

These comparison rules MUST be used for all JSON string comparisons except in cases where the definition of the member explicitly calls out that a different comparison rule is to be used for that member value. Only the "typ" and "cty" member values defined in this specification do not use these comparison rules.

Some applications may include case-insensitive information in a case-sensitive value, such as including a DNS name as part of a "kid" (key

ID) value. In those cases, the application may need to define a convention for the canonical case to use for representing the case-insensitive portions, such as lowercasing them, if more than one party might need to produce the same value so that they can be compared. (However if all other parties consume whatever value the producing party emitted verbatim without attempting to compare it to an independently produced value, then the case used by the producer will not matter.)

Also, see the JSON security considerations in Section 10.12 and the Unicode security considerations in Section 10.13.

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", "kid", "x5u", "x5c", "x5t", and "x5t#S256" 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; however, if the only information used in the trust decision is a key, these parameters need not be integrity protected, since changing them in a way that causes a different key to be used will cause the validation to fail.

The producer 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.

Also, see Appendix D for notes on possible key selection algorithms.

7. Serializations

JWSs use one of two serializations: the JWS Compact Serialization or the JWS JSON Serialization. Applications using this specification need to specify what serialization and serialization features are used for that application. For instance, applications might specify that only the JWS JSON Serialization is used, that only JWS JSON

Serialization support for a single signature or MAC value is used, or that support for multiple signatures and/or MAC values is used. JWS implementations only need to implement the features needed for the applications they are designed to support.

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 and it provides no syntax to represent a JWS Unprotected Header value.

7.2. JWS JSON Serialization

The JWS JSON Serialization represents digitally signed or MACed content as a JSON object. This representation is neither optimized for compactness nor URL-safe.

Two closely related syntaxes are defined for the JWS JSON Serialization: a fully general syntax, with which content can be secured with more than one digital signature and/or MAC operation, and a flattened syntax, which is optimized for the single digital signature or MAC case.

7.2.1. General JWS JSON Serialization Syntax

The following members are defined for use in top-level JSON objects used for the fully general JWS JSON Serialization syntax:

payload

The "payload" member MUST be present and contain the value
BASE64URL(JWS Payload).

signatures

The "signatures" member value MUST be an array of JSON objects. Each object represents a signature or MAC over the JWS Payload and the JWS Protected Header.

The following members are defined for use in the JSON objects that are elements of the "signatures" array:

protected

The "protected" member MUST be present and contain the value `BASE64URL(UTF8(JWS Protected Header))` when the JWS Protected Header value is non-empty; otherwise, it MUST be absent. These Header Parameter values are integrity protected.

header

The "header" member MUST be present and contain the value JWS Unprotected Header when the JWS Unprotected Header value is non-empty; otherwise, it MUST be absent. This value is represented as an unencoded JSON object, rather than as a string. These Header Parameter values are not integrity protected.

signature

The "signature" member MUST be present and contain the value `BASE64URL(JWS Signature)`.

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.

Additional members can be present in both the JSON objects defined above; if not understood by implementations encountering them, they MUST be ignored.

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 represented in the "protected" member of the signature/MAC's array element, and (2) the JWS Unprotected Header in the "header" member of the signature/MAC's array element. The union of these sets of Header Parameters comprises the JOSE Header. The Header Parameter names in the two locations MUST be disjoint.

Each JWS Signature value is computed using the parameters of the corresponding JOSE 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.

In summary, the syntax of a JWS using the general 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>" }
  ]
}
```

See Appendix A.6 for an example JWS using the general JWS JSON Serialization syntax.

7.2.2. Flattened JWS JSON Serialization Syntax

The flattened JWS JSON Serialization syntax is based upon the general syntax, but flattens it, optimizing it for the single digital signature/MAC case. It flattens it by removing the "signatures" member and instead placing those members defined for use in the "signatures" array (the "protected", "header", and "signature" members) in the top-level JSON object (at the same level as the "payload" member).

The "signatures" member MUST NOT be present when using this syntax. Other than this syntax difference, JWS JSON Serialization objects using the flattened syntax are processed identically to those using the general syntax.

In summary, the syntax of a JWS using the flattened JWS JSON Serialization is as follows:

```
{
  "payload": "<payload contents>",
  "protected": "<integrity-protected header contents>",
  "header": "<non-integrity-protected header contents>",
  "signature": "<signature contents>"
}
```

See Appendix A.7 for an example JWS using the flattened JWS JSON Serialization syntax.

8. TLS Requirements

Implementations supporting the "jku" and/or "x5u" Header Parameters MUST support TLS. Which TLS 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.

To protect against information disclosure and tampering, confidentiality protection MUST be applied using TLS with a ciphersuite that provides confidentiality and integrity protection. See current publications by the IETF TLS working group, including RFC 6176 [RFC6176], for guidance on the ciphersuites currently considered to be appropriate for use. Also, see Recommendations for Secure Use of TLS and DTLS [I-D.ietf-uta-tls-bcp] for recommendations on improving the security of software and services using TLS.

Whenever TLS is used, the identity of the service provider encoded in the TLS server certificate MUST be verified using the procedures described in Section 6 of RFC 6125 [RFC6125].

9. IANA Considerations

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

Values are registered on a Specification Required [RFC5226] basis after a three-week review period on the jose-reg-review@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 jose-reg-review@ietf.org mailing list for review and comment, with an appropriate subject (e.g., "Request to register header parameter: example").

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@ietf.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 description is clear.

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).

[[Note to the RFC Editor and IANA: Pearl Liang of ICANN had requested that the draft supply the following proposed registry description information. It is to be used for all registries established by this specification.

- o Protocol Category: JSON Object Signing and Encryption (JOSE)
- o Registry Location: <http://www.iana.org/assignments/jose>
- o Webpage Title: (same as the protocol category)
- o Registry Name: (same as the section title, but excluding the word "Registry", for example "JSON Web Signature and Encryption Header Parameters")

]]

9.1. JSON Web Signature and Encryption Header Parameters Registry

This specification establishes the IANA JSON Web Signature and Encryption Header Parameters registry for Header Parameter names. The registry records the Header Parameter name and a reference to the specification that defines it. The same Header Parameter name can 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.

9.1.1.1. Registration Template

Header Parameter Name:

The name requested (e.g., "kid"). 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 Description:

Brief description of the Header Parameter (e.g., "Key ID").

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.

9.1.1.2. 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 Description: Algorithm
- 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 Description: JWK Set URL
- 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 Description: JSON Web Key
- 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: "kid"
- o Header Parameter Description: Key ID
- 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: "x5u"
- o Header Parameter Description: X.509 URL
- 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 Description: X.509 Certificate Chain
- 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: "x5t"
- o Header Parameter Description: X.509 Certificate SHA-1 Thumbprint
- 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: "x5t#S256"
- o Header Parameter Description: X.509 Certificate SHA-256 Thumbprint
- 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: "typ"
- o Header Parameter Description: Type
- 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: "cty"
- o Header Parameter Description: Content Type
- o Header Parameter Usage Location(s): JWS
- o Change Controller: IESG

- o Specification Document(s): Section 4.1.10 of [[this document]]
- o Header Parameter Name: "crit"
- o Header Parameter Description: Critical
- o Header Parameter Usage Location(s): JWS
- o Change Controller: IESG
- o Specification Document(s): Section 4.1.11 of [[this document]]

9.2. Media Type Registration

9.2.1. Registry Contents

This specification registers the "application/jose" Media Type [RFC2046] in the MIME Media Types registry [IANA.MediaTypes] in the manner described in RFC 6838 [RFC6838], which can be used to indicate that the content is a JWS or JWE 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 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) each separated from the next by a single period ('.') character.
- 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, Android, Windows Azure, Xbox One, Amazon Web Services, and numerous others that use JWTs
- o Fragment identifier considerations: n/a
- 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 Provisional registration? No

- 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 Fragment identifier considerations: n/a
- 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 Provisional registration? No

10. Security Considerations

All of the security issues that are pertinent to any cryptographic application must be addressed by JWS/JWE/JWK agents. Among these issues are protecting the user's asymmetric private and symmetric secret keys and employing countermeasures to various attacks.

All the security considerations in XML DSIG 2.0 [W3C.NOTE-xmldsig-core2-20130411], 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.NOTE-xmldsig-bestpractices-20130411] also apply to this specification, other than those that are XML specific.

10.1. Key Entropy and Random Values

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.

Implementations must randomly generate public/private key pairs, message authentication (MAC) keys, and padding values. The use of inadequate pseudo-random number generators (PRNGs) to generate cryptographic keys can result in little or no security. An attacker

may find it much easier to reproduce the PRNG environment that produced the keys, searching the resulting small set of possibilities, rather than brute force searching the whole key space. The generation of quality random numbers is difficult. RFC 4086 [RFC4086] offers important guidance in this area.

10.2. Key Protection

Implementations must protect the signer's private key. Compromise of the signer's private key permits an attacker to masquerade as the signer.

Implementations must protect the message authentication (MAC) key. Compromise of the MAC key may result in undetectable modification of the authenticated content.

10.3. Key Origin Authentication

The key management technique employed to obtain public keys must authenticate the origin of the key; otherwise, it is unknown what party signed the message.

Likewise, the key management technique employed to distribute MAC keys must provide data origin authentication; otherwise, the contents are delivered with integrity from an unknown source.

10.4. Cryptographic Agility

See Section 8.1 of [JWA] for security considerations on cryptographic agility.

10.5. Differences between Digital Signatures and MACs

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

Both signatures and MACs provide for integrity checking -- verifying that the message has not been modified since the integrity value was computed. However, MACs provide for origination identification only under specific circumstances. It can normally be assumed that a private key used for a signature is only in the hands of a single entity (although perhaps a distributed entity, in the case of replicated servers); however, a MAC key needs to be in the hands of all the entities that use it for integrity computation and checking. Validation of a MAC only provides corroboration that the message was

generated by one of the parties that knows the symmetric MAC key. This means that origination can only be determined if a MAC key is known only to two entities and the recipient knows that it did not create the message. MAC validation cannot be used to prove origination to a third party.

10.6. Algorithm Validation

The digital signature representations for some algorithms include information about the algorithm used inside the signature value. For instance, signatures produced with RSASSA-PKCS-v1_5 [RFC3447] encode the hash function used and many libraries actually use the hash algorithm specified inside the signature when validating the signature. When using such libraries, as part of the algorithm validation performed, implementations MUST ensure that the algorithm information encoded in the signature corresponds to that specified with the "alg" Header Parameter. If this is not done, an attacker could claim to have used a strong hash algorithm while actually using a weak one represented in the signature value.

10.7. Algorithm Protection

In some usages of JWS, there is a risk of algorithm substitution attacks, in which an attacker can use an existing digital signature value with a different signature algorithm to make it appear that a signer has signed something that it has not. These attacks have been discussed in detail in the context of CMS [RFC6211]. This risk arises when all of the following are true:

- o Verifiers of a signature support multiple algorithms.
- o Given an existing signature, an attacker can find another payload that produces the same signature value with a different algorithm.
- o The payload crafted by the attacker is valid in the application context.

There are several ways for an application to mitigate algorithm substitution attacks:

- o Use only digital signature algorithms that are not vulnerable to substitution attacks. Substitution attacks are only feasible if an attacker can compute pre-images for a hash function accepted by the recipient. All JWA-defined signature algorithms use SHA-2 hashes, for which there are no known pre-image attacks, as of the time of this writing.

- o Require that the "alg" Header Parameter be carried in the protected header. (This is always the case when using the JWS Compact Serialization and is the approach taken by CMS [RFC6211].)
- o Include a field containing the algorithm in the application payload, and require that it be matched with the "alg" Header Parameter during verification. (This is the approach taken by PKIX [RFC5280].)

10.8. Chosen Plaintext Attacks

Creators of JWSs should not allow third parties to insert arbitrary content into the message without adding entropy not controlled by the third party.

10.9. Timing Attacks

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.

10.10. Replay Protection

While not directly in scope for this specification, note that applications using JWS (or JWE) objects can thwart replay attacks by including a unique message identifier as integrity protected content in the JWS (or JWE) message and having the recipient verify that the message has not been previously received or acted upon.

10.11. SHA-1 Certificate Thumbprints

A SHA-1 hash is used when computing "x5t" (X.509 Certificate SHA-1 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.

Alternatively, the "x5t#S256" (X.509 Certificate SHA-256 Thumbprint) Header Parameter could be used instead of "x5t". However, at the time of this writing, no development platform is known to support SHA-256 certificate thumbprints.

10.12. JSON Security Considerations

Strict JSON [RFC7159] validation is a security requirement. If malformed JSON is received, then the intent of the producer is impossible to reliably discern. Ambiguous and potentially exploitable situations could arise if the JSON parser used does not reject malformed JSON syntax. In particular, any JSON inputs not conforming to the JSON-text syntax defined in RFC 7159 input MUST be rejected in their entirety by JSON parsers.

Section 4 of the JSON Data Interchange Format specification [RFC7159] states "The names within an object SHOULD be unique", whereas this specification states that "Header Parameter names within this object MUST be unique; JWS parsers 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]". Thus, this specification requires that the Section 4 "SHOULD" be treated as a "MUST" by producers and that it be either treated as a "MUST" or in the manner specified in ECMA Script 5.1 by consumers. 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-text object followed by the extra characters "ABCD". Implementations MUST consider JWSs containing such input to be invalid.

10.13. 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 Section 8.3 of RFC 7159 [RFC7159]). 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.

11. References

11.1. Normative References

[ECMAScript]

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

[IANA.MediaTypees]

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), January 2015.

[JWK]

Jones, M., "JSON Web Key (JWK)", draft-ietf-jose-json-web-key (work in progress), January 2015.

[RFC20]

Cerf, V., "ASCII format for Network Interchange", RFC 20, October 1969.

[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.

[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.

- [RFC4648] Josefsson, S., "The Base16, Base32, and Base64 Data Encodings", RFC 4648, October 2006.
- [RFC4945] Korver, B., "The Internet IP Security PKI Profile of IKEv1/ISAKMP, IKEv2, and PKIX", RFC 4945, August 2007.
- [RFC4949] Shirey, R., "Internet Security Glossary, Version 2", RFC 4949, August 2007.
- [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.
- [RFC6176] Turner, S. and T. Polk, "Prohibiting Secure Sockets Layer (SSL) Version 2.0", RFC 6176, March 2011.
- [RFC7159] Bray, T., "The JavaScript Object Notation (JSON) Data Interchange Format", RFC 7159, March 2014.
- [UNICODE] The Unicode Consortium, "The Unicode Standard", 1991-, <<http://www.unicode.org/versions/latest/>>.

11.2. Informative References

- [CanvasApp] Facebook, "Canvas Applications", 2010.
- [I-D.ietf-uta-tls-bcp] Sheffer, Y., Holz, R., and P. Saint-Andre, "Recommendations for Secure Use of TLS and DTLS", draft-ietf-uta-tls-bcp-08 (work in progress), December 2014.
- [JSS] Bradley, J. and N. Sakimura (editor), "JSON Simple Sign", September 2010.

- [JWE] Jones, M. and J. Hildebrand, "JSON Web Encryption (JWE)", draft-ietf-jose-json-web-encryption (work in progress), January 2015.
- [JWT] Jones, M., Bradley, J., and N. Sakimura, "JSON Web Token (JWT)", draft-ietf-oauth-json-web-token (work in progress), January 2015.
- [MagicSignatures]
Panzer (editor), J., Laurie, B., and D. Balfanz, "Magic Signatures", January 2011.
- [RFC2104] Krawczyk, H., Bellare, M., and R. Canetti, "HMAC: Keyed-Hashing for Message Authentication", RFC 2104, February 1997.
- [RFC3447] Jonsson, J. and B. Kaliski, "Public-Key Cryptography Standards (PKCS) #1: RSA Cryptography Specifications Version 2.1", RFC 3447, February 2003.
- [RFC4086] Eastlake, D., Schiller, J., and S. Crocker, "Randomness Requirements for Security", BCP 106, RFC 4086, June 2005.
- [RFC4122] Leach, P., Mealling, M., and R. Salz, "A Universally Unique Identifier (UUID) URN Namespace", RFC 4122, July 2005.
- [RFC5226] Narten, T. and H. Alvestrand, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 5226, May 2008.
- [RFC6211] Schaad, J., "Cryptographic Message Syntax (CMS) Algorithm Identifier Protection Attribute", RFC 6211, April 2011.
- [RFC6838] Freed, N., Klensin, J., and T. Hansen, "Media Type Specifications and Registration Procedures", BCP 13, RFC 6838, January 2013.
- [SHS] National Institute of Standards and Technology, "Secure Hash Standard (SHS)", FIPS PUB 180-4, March 2012.
- [W3C.NOTE-xmlsig-bestpractices-20130411]
Hirsch, F. and P. Datta, "XML Signature Best Practices", World Wide Web Consortium Note NOTE-xmlsig-bestpractices-20130411, April 2013, <<http://www.w3.org/TR/2013/NOTE-xmlsig-bestpractices-20130411/>>.
- [W3C.NOTE-xmlsig-core2-20130411]

Eastlake, D., Reagle, J., Solo, D., Hirsch, F., Roessler, T., Yiu, K., Datta, P., and S. Cantor, "XML Signature Syntax and Processing Version 2.0", World Wide Web Consortium Note NOTE-xmlsig-core2-20130411, April 2013, <<http://www.w3.org/TR/2013/NOTE-xmlsig-core2-20130411/>>.

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" }
```

To remove potential ambiguities in the representation of the JSON object above, the actual octet sequence representing UTF8(JWS Protected Header) used in this example is also included below. (Note that ambiguities can arise due to differing platform representations of line breaks (CRLF versus LF), differing spacing at the beginning and ends of lines, whether the last line has a terminating line break or not, and other causes. In the representation used in this example, the first line has no leading or trailing spaces, a CRLF line break (13, 10) occurs between the first and second lines, the second line has one leading space (32) and no trailing spaces, and the last line does not have a terminating line break.) The octets representing UTF8(JWS Protected Header) in this example (using JSON array notation) 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 used in this example for 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
cGxlLmNvbS9pc19yb290Ijp0cnVlfQ
```

Combining these as BASE64URL(UTF8(JWS Protected Header)) || '.' || BASE64URL(JWS Payload) gives this string (with line breaks for display purposes only):

```
eyJ0eXAiOiJKV1QiLA0KICJhbGciOiJIUzI1NiJ9
.
eyJpc3MiOiJqb2UiLA0KICJleHAiOjEzMDA4MTkzODAsDQogImh0dHA6Ly9leGFT
cGxlLmNvbS9pc19yb290Ijp0cnVlfQ
```

The resulting JWS Signing Input value, which is the ASCII representation of above string, is the following octet sequence (using JSON array notation):

```
[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

within values 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_wW1gFWFOEjXk
```

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  
cGxlLnNvbS9pc19yb290Ijp0cnVlfQ  
.  
dBjftJeZ4CVP-mB92K27uhbUJU1plr_wW1gFWFOEjXk
```

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 (which is the initial substring of the JWS Compact Serialization representation up until but not including the second period character) as input to the HMAC SHA-256 function and then taking the output and determining if it matches the JWS Signature (which is base64url decoded from the value encoded in the JWS representation). 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 example (using JSON array notation) 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  
cGxlLnNvbS9pc19yb290Ijp0cnVlfQ
```

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 within values for display purposes only):

```
{ "kty": "RSA",
  "n": "ofgWCuLjybRlzo0tZWJjNiusfb4p4fAkd_wWJcyQoTbj9k0l8W26mPddx
HmfHQp-Vaw-4qPCJrcS2mJPMEzPlPt0Bm4d4QlL-yRT-SFd2lZS-pCgNMs
DlW_YpRPEwOWvG6b32690r2jz47soMzo9wGzjb_70Mg0LOL-bSf63kpaSH
SXndS5z5rexMdbBYUsLA9e-KXBdQOS-UTo7WTBEMa2R2CapHg665xsmtDV
MTBQY4u4DZlxvb3qCo5ZwKh9kG4LT6_I5IhlJH7aGhyxXFvUK-DWNmoudF8
NAco9_h9iaGNj8q2ethFkMLs91kzk2PAcDTW9gb54h4FRWyuXpoQ",
  "e": "AQAB",
  "d": "Eq5xpGnNCivDflJsRQBxHx1hDr1k6Ulwe2JZD50LpXyWPEAEp88vLNO97I
jla7_GQ5sLKMgvfTeXZx9SE-7YwVol2NXOoAJe46sui395IW_GO-pWJl00
BkTGoVEn2bKVRUCgu-GjBVAyLU6f3l9kJfFNS3E0QbVdxzubSu3Mkqzjkn
439X0M_V5l9gfpRLI9JYanrC4D4qAdGcopV_0ZHHZQlBjudU2QvXt4ehNYT
CBR6XCLQUShb1juU0lZdiYoFaFQT5Tw8bGUl_x_jTj3ccPDVZFD9pIuhLh
BOneufuBiB4cs98l2SR_RQyGWSeWjnczT0QU9lplDhOVRuOopznQ",
  "p": "4BzEEotIpmVdVEZNCqS7baC4crd0pqnRH_5IB3jw3bcxGn6QLvnEtfdUdi
YrqBdss1l58BQ3KhooKeQTa9AB0Hw_Py5PJdTJNPY8cQn7ouZ2KKDcmnPG
BY5t7yLclQlQ5xHdwWlVhvKn-nXqhJTBgIPgtldC-KDV5z-y2XDwGUc",
  "q": "uQPEfgmVtjL0Uyyx88GZFF1fOunH3-7cepKmtH4pxhtCoHqpWmT8YAmZxa
ewHgHAjLYsp1ZSe7zFYHj7C6ul7TjeLQeZD_YwD66t62wDmpe_HlB-TnBA
-njbgIfIsRLtXlnDzQkv5dTltRJl1BKBBypeeF6689rjcJIDEz9RWdc",
  "dp": "BwKfV3Akq5_MFZDFZCnW-wzl-CCo83WoZvnlQwCTeDv8uzluRSnm71l3Q
CLdhrqE2e9YkxvuxdBfPT_PI7Yz-FOKnulR6HsJeDCjn12Sk3vmAktV2zb
34MCdy7cpdTh_YVr7tss2u6vneTwrA86rZtu5Mbr1ClXsmvKxHQAdYo0",
  "dq": "h_96-mKlR_7glhsum8ldZxjTnYynPbZpHzizjeeHcXYSXaaMwkOlODsWa
7I9xXDoRwbKgB719rrmI2oKr6N3Do9U0ajaHF-NKJnwgjMd2w9cjz3_-ky
NlxAr2v4IKhGNpmM5iIgOS1VZnOZ68m6_pbLBSp3nssTdlqvdt0tiTHU",
  "qi": "IYd7DHOhrWvxkwPQSRM2tOgrjbcrfvtQJipd-DlcxyVuuM9sQLdgjV2o
y26F0EmpScGLq2MowX7fhd_QJQ3ydy5cY7YIBi87w93IKLEdfnbJtoOPLU
W0ITrJReOgolcq9SbsxYawBgfp_gh6A5603k2-ZQwVK0JKShuLFkuQ3U"
}
```

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_O8W2Fp5jujGbds9uJdbF9CUAr7tldnZcAcQjbKBYNX4
BAynRFdiuB--f_nZLgrnbyTyWzO75vRK5h6xBarLIARNPvkSjtQBMHlb1L07Qe7K
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
cGxlLnNvbs9pc19yb290Ijpb0cnVlfiQ
.
cC4hiUPoj9Eetdgtv3hF80EGrhuB__dzERat0XF9g2VtQgr9PJbu3XOiZj5RZmh7
AAuHIm4Bh-0Qc_lF5YKt_O8W2Fp5jujGbds9uJdbF9CUAr7tldnZcAcQjbKBYNX4
BAynRFdiuB--f_nZLgrnbyTyWzO75vRK5h6xBarLIARNPvkSjtQBMHlb1L07Qe7K
0GarZRmB_eSN9383LcOLn6_dO--xil2jzDwusC-eOkHWesqtFZESc6BfI7noOPqv
hJlphCnvWh6IeYI2w9QOYEUIpUTI8np6LbgGY9Fs98rqVt5AXLIhWkWywlvmtVrB
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 bit different from the previous

example. We pass the public key (n, e), the JWS Signature (which is base64url decoded from the value encoded in the JWS representation), and the JWS Signing Input (which is the initial substring of the JWS Compact Serialization representation up until but not including the second period character) 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 example (using JSON array notation) 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):

```
eyJhbGciOiJFUzI1NiJ9  
.  
eyJpc3MiOiJqb2UiLA0KICJleHAiOjEzMDA4MTkzODAsDQogImh0dHA6Ly9leGft  
cGxlLnNvbS9pc19yb290Ijp0cnVlfQ
```

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-F4GawxaepmXFCgfTjDxw5djsxLa8ISlSA
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
cGxlLnNvbS9pc19yb290Ijp0cnVlfQ
.
DtEhU3ljbEg8L38VWafUAqOyKAM6-Xx-F4GawxaepmXFCgftjDxw5djxLa8ISlSA
pmWQxfKTUJqPP3-Kg6NU1Q
```

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 bit different from the previous examples. We need to split the 64 member octet sequence of the JWS Signature (which is base64url decoded from the value encoded in the JWS representation) into two 32 octet sequences, the first representing R and the second S. We then pass the public key (x, y), the signature (R, S), and the JWS Signing Input (which is the initial substring of the JWS Compact Serialization representation up until but not including the second period character) 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 example (using JSON array notation) 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:

eyJhbGciOiJIJFZUZXMiJ9.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 within values for display purposes only):

```
{ "kty": "EC",
  "crv": "P-521",
  "x": "AekpBQ8ST8a8VcfVOTNl353vSrDCLLJXmPk06wTjxrrjcBpXp5EOnYG_
NjFZ6OvLFVljSfS9tsz4qUxcWceqwQGk",
  "y": "ADSmRA43ZlDSNx_RvcLI87cdL07l6jQyyBXMoxVg_l2Th-x3S1WDhjDl
y79ajL4Kkd0AZMaZmh9ubmf63e3kyMj2",
  "d": "AY5pb7A0UFiB3RELSd64fTLOSv_jazdF7fLYyuTw8lOfRhWg6Y6rUrPA
xerEzgdRhajnu0ferB0d53vM9mE15j2C"
}
```

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-1192l60hp9wAHZfVJbLfD_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-1192l60hp9wAHZfVJbLfD_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 this JWS Signature is very 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 representing R and the second S. We then pass the public key (x, y), the signature (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 Unsecured JWS

The following example JWS Protected Header declares that the encoded object is an Unsecured JWS:

```
{"alg": "none"}
```

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

```
eyJhbGciOiJub251In0
```

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 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):

```
eyJhbGciOiJub251In0  
.  
eyJpc3MiOiJqb2UiLA0KICJleHAiOjEzMDA4MTkzODAsDQogImh0dHA6Ly9leGft  
cGxlLnNvbS9pc19yb290Ijp0cnVlfQ  
.
```

A.6. Example JWS using General JWS JSON Serialization

This section contains an example using the general JWS JSON Serialization syntax. 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  
cGxlLnNvbS9pc19yb290Ijp0cnVlfQ
```

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 JOSE Header Values

Combining the protected and unprotected header values supplied, the JOSE 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 JWS JSON Serialization for these values is as follows (with line breaks within values for display purposes only):

```
{
  "payload":
    "eyJpc3MiOiJqb2UiLA0KICJleHAiOjEzMDA4MTkzODAsDQogImh0dHA6Ly9leGFtcGxlLnNvbS9pc19yb290Ijpb0cnVlfiQ",
  "signatures": [
    { "protected": "eyJhbGciOiJSUzI1NiJ9",
      "header":
        { "kid": "2010-12-29" },
      "signature":
        "cC4hiUPoj9Eetdgtv3hF80EGrhuB__dzERat0XF9g2VtQgr9PJbu3XOiZj5RZmh7AAuHIm4Bh-0Qc_lF5YKt_08W2Fp5jujGbds9uJdbF9CUAr7t1dnZcAcQjbKBYNX4BAynRFdiuB--f_nZLgrnbyTyWz075vRK5h6xBArLIARNPvkSjtQBMH1b1L07Qe7K0GarZRmB_eSN9383LcOLn6_d0--xi12jzDwusC-eOkHWesqtFZESc6BfI7noOPqvhJlphCnvWh6IeYI2w9QOYEUpUTI8np6LbgGY9Fs98rqVt5AXLIhWkWywlVmtVrBp0igcN_IoypGlUPQGe77Rw" },
    { "protected": "eyJhbGciOiJFUzI1NiJ9",
      "header":
        { "kid": "e9bc097a-ce51-4036-9562-d2ade882db0d" },
      "signature":
        "DtEhU3ljbEg8L38VWafUAqOyKAM6-Xx-F4GawxaepmXFCgftTjDxw5djsxLa8ISlSApmWQxfKTUJqPP3-Kg6NUlQ" } ]
}
```

A.7. Example JWS using Flattened JWS JSON Serialization

This section contains an example using the flattened JWS JSON Serialization syntax. This example demonstrates the capability for conveying a single digital signature or MAC in a flattened JSON structure.

The values in this example are the same as those in the second signature of the previous example in Appendix A.6.

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

```
{
  "payload":
    "eyJpc3MiOiJqb2UiLA0KICJleHAiOjEzMDA4MTkzODAsDQogImh0dHA6Ly9leGFtcGxlLmNvbS9pc19yb290Ijp0cnVlfQ",
  "protected": "eyJhbGciOiJFUzI1NiJ9",
  "header":
    { "kid": "e9bc097a-ce51-4036-9562-d2ade882db0d" },
  "signature":
    "DtEhU31jbEg8L38VWafUAqOyKAM6-Xx-F4GawxaepmXFCgfTjDxw5djxLa8ISlSApmWQxfKTUJqPP3-Kg6NU1Q"
}
```

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 (with line breaks within values for display purposes only):

```
[ "MIIE3jCCA8agAwIBAgICAwEwDQYJKoZIhvcNAQEFBQAwYzELMAkGA1UEBhMCVVMxITAfBgNVBAoTGFROZSBHbyBEYWRkeSBHcm91cCwgSW5jLjExMC8GA1UECXMOR28gRGFkZHKgQ2xhc3MgMiBDZXJ0aWZpY2F0aW9uIEF1dGhvcml0eTAeFw0wNjExMTYwMTU0MzdaFw0yNjExMTYwMTU0MzdaMIHKMQswCQYDVQQGEwJVUzEQMA4GA1UECBMHQXJpem9uYTETMBEGA1UEBxMKU2NvdHRzZGFsZTEaMBGGA1UEChMRR29EYWRkeS5jb20sIEluYy4xMzAxBgNVBAsTKmh0dHA6Ly9jZXJ0aWZpY2F0ZXMuZ29kYWRkeS5jb20vcmlvbmVwb3NpdG9yeTEwMC4GA1UEAxMnR28gRGFkZHKgU2VjdXJlIENlc nRpZmljYXRpb24gQXV0aG9yaXR5MREwDwYDVQQFEwgnZk2OTI4NzCCASIwDQYJKoZIhvcNAQEBBQADggEPADCCAQoCggEBAMQtlRWMnCMZ7DI161+4WQFapmGBWTTwY6vj3D3HKrjJM9N55DrtPDAjhI6zMBSSofDPZVUBJ7fmd0LJR4h3mUpfjWoqVTr9vcyOdQmVZWt7/v+WibXnvQAJYwqDL1CBM6nPWT27oDyqu9SoWlm2r4arV3aL GbqGmu75RpRSgAvSMeYddi5Kcju+GZtCpyz8/x4fKL4o/Klw/O5epHBp+YlLpyo7RJlbmr2EkRTcDCVw5wrWCs9CHRK8r5RsL+H0EwnWGu1NcWdrxcx+AuP7q2BNgWJCJjPOq8lh8BJ6qf9Z/dFjpfMFDniNoW1fho3/Rb2cRGadDAW/hOUoz+EDU8CAW EAAaOCATiWggEuMB0GA1UdDgQWBBT9rGEyk2xFluLuhV+auud2mWjm5zAfBgNVHSMEGDAGBTsXLDskdRMEXGzYcs9of7dqGrU4zASBgNVHRMBAf8ECDAGAQH/AgEAMDMGCCsGAQUFBwEBBCcwJTAjBggrBgEFBQcwAYYXaHR0cDovL29jc3AuZ29kYWRkeS5jb20wRgYDVDR0fBD8wPTA7oDmgN4Y1aHR0cDovL2N1cnRpZmljYXRlc y5nb2RhZGR5LmNvbS9yZXBvc2l0b3J5L2dkcm9vdC5jcmwwSwYDVDR0gBEQwQjBAbGRVHSAAMDgwNgYIKwYBBQUHAgEWMh0dHA6Ly9jZXJ0aWZpY2F0ZXMuZ29kYWRkeS5jb20vcmlvbmVwb3NpdG9yeTAOBgNVHQ8BAf8EBAMCAQYwDQYJKoZIhvcNAQEFBQADggEBBANKGwOy9+aG2Z+5mC6IGORQjhVyrEp0lVPLN8tESe8HkGsz2Zbw1FalEzAFPIUyIXvJxwqoJKSQ3kbTJSMUA2fCENZvD117esyfxVgqwcSeIaha86ykRvOe5GPLL5CkKSkB2XIsKd83ASe8T+5o0yGPwLPk9Qnt0hCqU7S+8MxZC9Y71hyVJEnfuz9p0iRFEUOOjzv2kWzRaJBydTXRE4+uXR21aITVSzGh60lmawGhId/dQb8vxRMDsxuxN89txJx9OjxUUAiKEngHUuHqDTMBqLdElrRhjZkAzVvb3du6/KFUJheqwnTrZEjYx8WnM25sgVjOuH0aBsXBTWVU+4=",
  "MIIE+zCCBGsgAwIBAgICAQ0wDQYJKoZIhvcNAQEFBQAwgbsxJDAiBgNVBACGTGlZ
```

hbG1DZXJ0IFZhbG1kYXRpb24gTmV0d29yazEXMBUGA1UEChMOVmFsaUNlcnQsIEluYy4xNTAzBgNVBAsTLFZhbG1DZXJ0IENsYXNzIDIGUG9saWN5IFZhbG1kYXRpb24gQXV0aG9yaXR5MSEwHwYDVQQDExhodHRwOi8vd3d3LnZhbG1jZXJ0LmNvbS8xIDAeBgkqhkiG9w0BCQEWEluZm9AdmFsaWNlcnQuY29tMB4XDTA0MDYyOTE3MDYyMFoXDTI0MDYyOTE3MDYyMFowYzELMAKGA1UEBhMCVVMxITAFBgNVBAoTGFRoZS BHbyBEYWRkeSBHcm91cCwgSW5jLjJEXMc8GA1UECXMOR28gRGFkZHKgQ2xhc3MgMiBDZXJ0aWZpY2F0aW9uIEFldGhvcml0eTCCASAwDQYJKoZIhvcNAQEBBQADggENADCCAQgCggEBAN6dl+pXGEmhW+vXX0iG6r7d/+TvZxz0ZWizV3GgXne77ZtJ6XCAPVYYYwhv2vLM0D9/AlQiVBDYsoHUWU9S3/Hd8M+eKsa7Ugay9qK7HFih7Eux6wwdhFJ2+qN1j3hybX2C32qRe3H3I2TqYXP2WYktsqbl2i/ojgC95/5Y0V4evLOtXiEqITLdiOr18SPaAIBQi2XKVLOARFmR6jYGB0xUGlcmIbYsUfbl8aQr4CUWwo riMYavx4A61nf4DD+qta/KFApMoZFv6yyO9ecw3ud72a9nmYvLEHZ6IVDd2gWMZEewo+YihfukeEHU1jPEX44dMX4/7VpkI+EdOqXG68CAQ0jggHhMIIB3TAdBgNVHQ4EFgQU0sSw0pHUTBFxs2HLPaH+3ahq1OMwgdIGA1UdIwSByjCBx6GBwaSBvjCBuzEkMCIGA1UEBxMbVmfSaUNlcnQgVmFsaWRhdGlvbiBOZXR3b3JrMRcwFQYDVQQKEw5WYXpQ2VydCwgSW5jLjE1MDMGA1UECXMsVmFsaUNlcnQgQ2xhc3MgMiBQb2xpyY3kgVmFsaWRhdGlvbiBBdXRob3JpdHkxITAFBgNVBAMTGgh0dHA6Ly93d3cudmFsaWNlcnQuY29tLzEgMB4GCSqGSIb3DQEJARYRaW5mb0B2YWxpY2VydC5jb22CAQEwDwYDVR0TAQH/BAUwAwEB/zAzBggrBgEFBQcBAQQnMCUwIwYIKwYBBQUHMAGF2h0dHA6Ly9vY3NwLmdvZGFkZHKuY29tMEQGA1UdHwQ9MDswOaA3oDWGM2h0dHA6Ly9jZXJ0aWZpY2F0ZXMuZ29kYWRkeS5jb20vcmluZm93NpdG9yeS9yb290LmNybDBLBG9NVHSAERDBCMEAGBFUdIAAwODA2BggrBgEFBQcCARYqaHR0cDovL2N1cnRpZmljYXRlcY5nb2RhZGR5LmNvbS9yZXBvc2l0b3J5MA4GA1UdDwEB/wQEAwIBBjANBgkqhkiG9w0BAQUFAAOBgQC1QPmnHfbq/qQaQlpe9xXUuUaJwL6e4+PrxeNYiY+SnleocSxI0YGyeR+sBjUZsE4OWBsUs5iB0QQeyAfJg594RAoYC5jcdnplDQ1tgMQLARzLrUc+cb53S8wGd9D0VmsfSxOaFIqII6hR8INMqzW/Rn453HWkrugp++85j09VZw==",

"MIIC5zCCALACAQEwDQYJKoZIhvcNAQEFBQAwwbsxJDAiBgNVBACtG1ZhbG1DZXJ0IFZhbG1kYXRpb24gTmV0d29yazEXMBUGA1UEChMOVmFsaUNlcnQsIEluYy4xNTAzBgNVBAsTLFZhbG1DZXJ0IENsYXNzIDIGUG9saWN5IFZhbG1kYXRpb24gQXV0aG9yaXR5MSEwHwYDVQQDExhodHRwOi8vd3d3LnZhbG1jZXJ0LmNvbS8xIDAeBgkqhkiG9w0BCQEWEluZm9AdmFsaWNlcnQuY29tMB4XDTk5MDYyNjAwMTk1NFoXDTI0MDYyNjAwMTk1NFowgbsxJDAiBgNVBACtG1ZhbG1DZXJ0IFZhbG1kYXRpb24gTmV0d29yazEXMBUGA1UEChMOVmFsaUNlcnQsIEluYy4xNTAzBgNVBAsTLFZhbG1DZXJ0IENsYXNzIDIGUG9saWN5IFZhbG1kYXRpb24gQXV0aG9yaXR5MSEwHwYDVQQDExhodHRwOi8vd3d3LnZhbG1jZXJ0LmNvbS8xIDAeBgkqhkiG9w0BCQEWEluZm9AdmFsaWNlcnQuY29tMIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQDOOnHK5avIWZJV16vYdA757tn2VUDZZUcOBVXc65g2PFxTXdMwzzjsvUGJ7SVCCSRrC16zfn1SLUzmlNZ9WlmpZdRJEy0kTRxQb7XBhVQ7/nHk01xC+YDgkRoKWzk2Z/M/VXwbP7RfZHM047QSV4dk+NoS/zcnwbNDu+97bi5p9wIDAQABMA0GCSqGSIb3DQEBAQUAA4GBADt/UG9vUJSZSWI4OB9L+KXIPqeCgfYrx+jFzug6EILLGACOTb2oWH+heQClu+mNr0HZDzTuIYEZoDJJKPTEjlbVUjP9UNV+mWwD5MlM/Mtsq2azSiGM5bUMMJ4QssxsodyamEwCW/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. Notes on Key Selection

This appendix describes a set of possible algorithms for selecting the key to be used to validate the digital signature or MAC of a JWS or for selecting the key to be used to decrypt a JWE. This guidance describes a family of possible algorithms, rather than a single algorithm, because in different contexts, not all the sources of keys will be used, they can be tried in different orders, and sometimes not all the collected keys will be tried; hence, different algorithms will be used in different application contexts.

The steps below are described for illustration purposes only; specific applications can and are likely to use different algorithms or perform some of the steps in different orders. Specific applications will frequently have a much simpler method of determining the keys to use, as there may be one or two key selection methods that are profiled for the application's use. This appendix supplements the normative information on key location in Section 6.

These algorithms include the following steps. Note that the steps can be performed in any order and do not need to be treated as distinct. For example, keys can be tried as soon as they are found, rather than collecting all the keys before trying any.

1. Collect the set of potentially applicable keys. Sources of keys may include:
 - * Keys supplied by the application protocol being used.
 - * Keys referenced by the "jku" (JWK Set URL) Header Parameter.
 - * The key provided by the "jwk" (JSON Web Key) Header Parameter.
 - * The key referenced by the "x5u" (X.509 URL) Header Parameter.
 - * The key provided by the "x5c" (X.509 Certificate Chain) Header Parameter.
 - * Other applicable keys available to the application.

The order for collecting and trying keys from different key sources is typically application dependent. For example, frequently all keys from a one set of locations, such as local caches, will be tried before collecting and trying keys from other locations.

2. Filter the set of collected keys. For instance, some applications will use only keys referenced by "kid" (key ID) or

"x5t" (X.509 certificate SHA-1 thumbprint) parameters. If the application uses the "alg" (algorithm), "use" (public key use), or "key_ops" (key operations) parameters, keys with keys with inappropriate values of those parameters would be excluded. Additionally, keys might be filtered to include or exclude keys with certain other member values in an application specific manner. For some applications, no filtering will be applied.

3. Order the set of collected keys. For instance, keys referenced by "kid" (Key ID) or "x5t" (X.509 Certificate SHA-1 Thumbprint) parameters might be tried before keys with neither of these values. Likewise, keys with certain member values might be ordered before keys with other member values. For some applications, no ordering will be applied.
4. Make trust decisions about the keys. Signatures made with keys not meeting the application's trust criteria would not be accepted. Such criteria might include, but is not limited to the source of the key, whether the TLS certificate validates for keys retrieved from URLs, whether a key in an X.509 certificate is backed by a valid certificate chain, and other information known by the application.
5. Attempt signature or MAC validation for a JWS or decryption of a JWE with some or all of the collected and possibly filtered and/or ordered keys. A limit on the number of keys to be tried might be applied. This process will normally terminate following a successful validation or decryption.

Note that it is reasonable for some applications to perform signature or MAC validation prior to making a trust decision about a key, since keys for which the validation fails need no trust decision.

Appendix E. 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  
ZjTKVEIl0sDQogImh0dHA6Ly9leGFtcGxlLmNvbS9VTkRFRklORUQiOnRydWUNCn0.  
RkFJTA.
```

Appendix F. Detached Content

In some contexts, it is useful integrity protect content that is not itself contained in a JWS. One way to do this is create a JWS in the normal fashion using a representation of the content as the payload, but then delete the payload representation from the JWS, and send this modified object to the recipient, rather than the JWS. When using the JWS Compact Serialization, the deletion is accomplished by replacing the second field (which contains `BASE64URL(JWS Payload)`) value with the empty string; when using the JWS JSON Serialization, the deletion is accomplished by deleting the "payload" member. This method assumes that the recipient can reconstruct the exact payload used in the JWS. To use the modified object, the recipient reconstructs the JWS by re-inserting the payload representation into the modified object, and uses the resulting JWS in the usual manner. Note that this method needs no support from JWS libraries, as applications can use this method by modifying the inputs and outputs of standard JWS libraries.

Appendix G. 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, Alissa Cooper, Breno de Medeiros, Stephen Farrell, Dick Hardt, Joe Hildebrand, Jeff Hodges, Russ Housley, Edmund Jay, Tero Kivinen, Yaron Y. Goland, Ben Laurie, Ted Lemon, James Manger, Matt Miller, Kathleen Moriarty, Tony Nadalin, Hideki Nara, Axel Nennker, John Panzer, Ray Polk, Emmanuel Raviart, Eric Rescorla, Pete Resnick, Jim Schaad, Paul Tarjan, Hannes Tschofenig, and Sean Turner.

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

Appendix H. Document History

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

-41

- o Changed more instances of "reject" to "consider to be invalid".
- o Simplified the wording of a Message Signature or MAC Computation step.

-40

- o Clarified the definitions of UTF8(String) and ASCII(String).
- o Stated that line breaks are for display purposes only in places where this disclaimer was needed and missing.

-39

- o Updated the reference to draft-ietf-uta-tls-bcp.

-38

- o Replaced uses of the phrases "JWS object" and "JWE object" with "JWS" and "JWE".
- o Added member names to the JWS JSON Serialization Overview.
- o Applied other minor editorial improvements.

-37

- o Updated the TLS requirements language to only require implementations to support TLS when they support features using

TLS.

- o Updated the language about integrity protecting Header Parameters when used in a trust decision.
- o Restricted algorithm names to using only ASCII characters.
- o When describing actions taken as a result of validation failures, changed statements about rejecting the JWS to statements about considering the JWS to be invalid.
- o Added the CRT parameter values to example RSA private key representations.
- o Updated the example IANA registration request subject line.

-36

- o Defined a flattened JWS JSON Serialization syntax, which is optimized for the single digital signature or MAC case.
- o Clarified where white space and line breaks may occur in JSON objects by referencing Section 2 of RFC 7159.
- o Specified that registration reviews occur on the jose-reg-review@ietf.org mailing list.

-35

- o Addressed AppsDir reviews by Ray Polk.
- o Used real values for examples in the IANA Registration Template.

-34

- o Addressed IESG review comments by Alissa Cooper, Pete Resnick, Richard Barnes, Ted Lemon, and Stephen Farrell.
- o Addressed Gen-ART review comments by Russ Housley.
- o Referenced RFC 4945 for PEM certificate delimiter syntax.

-33

- o Noted that certificate thumbprints are also sometimes known as certificate fingerprints.

- o Added an informative reference to draft-ietf-uta-tls-bcp for recommendations on improving the security of software and services using TLS.
- o Changed the registration review period to three weeks.
- o Acknowledged additional contributors.

-32

- o Addressed Gen-ART review comments by Russ Housley.
- o Addressed secdir review comments by Tero Kivinen, Stephen Kent, and Scott Kelly.
- o Replaced the term Plaintext JWS with Unsecured JWS.

-31

- o Reworded the language about JWS implementations ignoring the "typ" and "cty" parameters, explicitly saying that their processing is performed by JWS applications.
- o Added additional guidance on ciphersuites currently considered to be appropriate for use, including a reference to a recent update by the TLS working group.

-30

- o Added subsection headings within the Overview section for the two serializations.
- o Added references and cleaned up the reference syntax in a few places.
- o Applied minor wording changes to the Security Considerations section and made other local editorial improvements.

-29

- o Replaced the terms JWS Header, JWE Header, and JWT Header with a single JOSE Header term defined in the JWS specification. This also enabled a single Header Parameter definition to be used and reduced other areas of duplication between specifications.

-28

- o Revised the introduction to the Security Considerations section. Also introduced additional subsection headings for security considerations items and also moved a security consideration item here from the JWA draft.
- o Added text about when applications typically would and would not use "typ" and "cty" header parameters.

-27

- o Added the "x5t#S256" (X.509 Certificate SHA-256 Thumbprint) header parameter.
- o Stated that any JSON inputs not conforming to the JSON-text syntax defined in RFC 7159 input MUST be rejected in their entirety.
- o Simplified the TLS requirements.

-26

- o Referenced Section 6 of RFC 6125 for TLS server certificate identity validation.
- o Described potential sources of ambiguity in representing the JSON objects used in the examples. The octets of the actual UTF-8 representations of the JSON objects used in the examples are included to remove these ambiguities.
- o Added a small amount of additional explanatory text to the signature validation examples to aid implementers.
- o Noted that octet sequences are depicted using JSON array notation.
- o Updated references, including to W3C specifications.

-25

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

-24

- o Updated the JSON reference to RFC 7159.

-23

- o Clarified that the base64url encoding includes no line breaks, white space, or other additional characters.

-22

- o Corrected RFC 2119 terminology usage.
- o Replaced references to draft-ietf-json-rfc4627bis with RFC 7158.

-21

- o Applied review comments to the appendix "Notes on Key Selection", addressing issue #93.
- o Changed some references from being normative to informative, addressing issue #90.
- o Applied review comments to the JSON Serialization section, addressing issue #121.

-20

- o Made terminology definitions more consistent, addressing issue #165.
- o Restructured the JSON Serialization section to call out the parameters used in hanging lists, addressing issue #121.
- o Described key filtering and refined other aspects of the text in the appendix "Notes on Key Selection", addressing issue #93.
- o Replaced references to RFC 4627 with draft-ietf-json-rfc4627bis, addressing issue #90.

-19

- o Added the appendix "Notes on Validation Key Selection", addressing issue #93.
- o Reordered the key selection parameters.

-18

- o Updated the mandatory-to-implement (MTI) language to say that applications using this specification need to specify what serialization and serialization features are used for that application, addressing issue #119.
- o Changes to address editorial and minor issues #25, #89, #97, #110, #114, #115, #116, #117, #120, and #184.

- o Added and used Header Parameter Description registry field.

-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-jws" 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 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
URI: <http://www.thread-safe.com/>

Nat Sakimura
Nomura Research Institute

Email: n-sakimura@nri.co.jp
URI: <http://nat.sakimura.org/>

JOSE Working Group
Internet-Draft
Intended status: Standards Track
Expires: June 30, 2013

M. Jones
Microsoft
December 27, 2012

JSON Web Encryption JSON Serialization (JWE-JS)
draft-jones-jose-jwe-json-serialization-04

Abstract

The JSON Web Encryption JSON Serialization (JWE-JS) is a means of representing encrypted content using JavaScript Object Notation (JSON) data structures. This specification describes a means of representing secured content as a JSON data object (as opposed to the JWE specification, which uses a compact serialization with a URL-safe representation). It enables the same content to be encrypted to multiple parties (unlike JWE). Cryptographic algorithms and identifiers used with this specification are described in the separate JSON Web Algorithms (JWA) specification. The JSON Serialization for related digital signature and MAC functionality is described in the separate JSON Web Signature JSON Serialization (JWS-JS) 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 June 30, 2013.

Copyright Notice

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

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents

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

Table of Contents

1. Introduction	3
1.1. Notational Conventions	3
2. Terminology	3
3. JSON Serialization	3
4. Example JWE-JS	5
5. IANA Considerations	6
6. Security Considerations	6
7. References	6
7.1. Normative References	6
7.2. Informative References	7
Appendix A. Acknowledgements	7
Appendix B. Open Issues	7
Appendix C. Document History	7
Author's Address	9

1. Introduction

The JSON Web Encryption JSON Serialization (JWE-JS) is a format for representing encrypted content as a JavaScript Object Notation (JSON) [RFC4627] object. It enables the same content to be encrypted to multiple parties (unlike JWE [JWE].) The encryption mechanisms are independent of the type of content being encrypted. Cryptographic algorithms and identifiers used with this specification are described in the separate JSON Web Algorithms (JWA) [JWA] specification. The JSON Serialization for related digital signature and MAC functionality is described in the separate JSON Web Signature JSON Serialization (JWS-JS) [JWS-JS] specification.

1.1. Notational Conventions

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

2. Terminology

This specification uses the same terminology as the JSON Web Encryption (JWE) [JWE] specification.

3. JSON Serialization

The JSON Serialization represents encrypted content as a JSON object with a "recipients" member containing an array of per-recipient information, an "initialization_vector" member containing a shared Encoded JWE Initialization Vector value, and a "ciphertext" member containing a shared Encoded JWE Ciphertext value. Each member of the "recipients" array is a JSON object with a "header" member containing an Encoded JWE Header value, an "encrypted_key" member containing an Encoded JWE Encrypted Key value, and an "integrity_value" member containing an Encoded JWE Integrity Value value.

Unlike the compact serialization used by JWEs, content using the JSON Serialization MAY be encrypted to more than one recipient. Each recipient requires:

- o a JWE Header value specifying the cryptographic parameters used to encrypt the JWE Encrypted Key to that recipient and the parameters used to encrypt the plaintext to produce the JWE Ciphertext; this is represented as an Encoded JWE Header value in the "header" member of an object in the "recipients" array.

- o a JWE Encrypted Key value used to encrypt the ciphertext; this is represented as an Encoded JWE Encrypted Key value in the "encrypted_key" member of the same object in the "recipients" array.
- o a JWE Integrity Value that ensures the integrity of the Ciphertext and the parameters used to create it; this is represented as an Encoded JWE Integrity Value value in the "integrity_value" member of the same object in the "recipients" array.

Therefore, the syntax is:

```
{ "recipients": [
  { "header": "<header 1 contents>",
    "encrypted_key": "<encrypted key 1 contents>",
    "integrity_value": "<integrity value 1 contents>" },
  ...
  { "header": "<header N contents>",
    "encrypted_key": "<encrypted key N contents>",
    "integrity_value": "<integrity value N contents>" } ],
  "initialization_vector": "<initialization vector contents>",
  "ciphertext": "<ciphertext contents>"
}
```

The contents of the Encoded JWE Header, Encoded JWE Encrypted Key, Encoded JWE Initialization Vector, Encoded JWE Ciphertext, and Encoded JWE Integrity Value values are exactly as specified in JSON Web Encryption (JWE) [JWE]. They are interpreted and validated in the same manner, with each corresponding "header", "encrypted_key", and "integrity_value" value being created and validated together.

Each JWE Encrypted Key value and the corresponding JWE Integrity Value are computed using the parameters of the corresponding JWE Header value in the same manner described in the JWE specification. This has the desirable result that each Encoded JWE Encrypted Key value in the "recipients" array and each Encoded JWE Integrity Value in the same array element are identical to the values that would have been computed for the same parameters in a JWE, as is the shared JWE Ciphertext value.

All recipients use the same JWE Ciphertext and JWE Initialization Vector values, resulting in potentially significant space savings if the message is large. Therefore, all header parameters that specify the treatment of the JWE Ciphertext 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 MUST be the same.

4. Example JWE-JS

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 first using the RSAES-PKCS1-V1_5 algorithm to encrypt the Content Master Key (CMK) and the second using RSAES OAEP to encrypt the CMK. The Plaintext is encrypted using the AES CBC algorithm and the same block encryption parameters to produce the common JWE Ciphertext value. The two Decoded JWE Header Segments used are:

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

and:

```
{ "alg": "RSA-OAEP", "enc": "A128CBC+HS256" }
```

The keys used for the first recipient are the same as those in Appendix A.2 of [JWE], as is the plaintext used. The asymmetric encryption key used for the second recipient is the same as that used in Appendix A.1 of [JWE]; the block encryption keys and parameters for the second recipient are the same as those for the first recipient (which must be the case, since the initialization vector and ciphertext are shared).

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

```

{
  "recipients": [
    {
      "header": {
        "eyJhbGciOiJSU0ExXzUiLCJlbnMiOiJBMTI4Q0JDK0hTMjU2In0",
        "encrypted_key":
          "ZmnlqWgjXyqwjr7cXHys8F79anIUI6J2UWdAyRQEcGBU-KPHsePM910_RoTDG
          ulIW40Dn0dvcdVEjPjCpPNibzWcMxDi131Ejeg-b8ViW5YX5oRdYdiR4gMSDD
          B3mbkInMNUFT-PK5CuZRnHB2rUK5fhPuF6XFqLLZCG5Q_rJm6Evex-XLcNQAj
          Na1-6CIU12Wj3mPExxw9vbnsQDU7B4BfmhdyiflLA7Ae5ZGoVR13A__yLPXxR
          jHFhpOeDp_adx8NyejF5cz9yDKULugNsDMdlHeJQOMGVLYaSZt3KP6aWNSqFA
          lPHDg-10ceuTEtq_vPE4-Gtev4N4K4Eudlj4Q",
        "integrity_value":
          "8LXqMd0JLGsxMaB5uoNaMpg7uUW_p40RlaZHCwMIyzk"
      },
      {
        "header": {
          "eyJhbGciOiJSU0EtT0FFUCIsImVuYyI6IkExMjhDQkMrSFMMyNTYifQ",
          "encrypted_key":
            "nxwnYB86zEvVRofSxnDuFAE9-Gi2JtCy5eMyYedowjfGlkoA-Y0JyfwWXE_EU
            vhf6WS_DM3a18You2Qsah3BvvSRPZ8-TNYX3_4QCEO-V8EVbFleGoJFs9ODmC
            cOiuM1lxLnSAYlwEDDnhwEkXv8o6MZEvh-msqTY6NyFGd6mhjpu9P4o2F2hOe
            Nt6FthcR4cNpAVSbydeEBSzsrp27nB-JwfmLjnSYQ0lJBwbguUJXHzyIJcQa7i
            43Vko02HkWTxBta0q5Zr_Jd7V2l-6HLYIuNc7fZH1DSJSTBTotcugumR5zNX_
            uxQyMoWoQ-SsQ7HxqrrFbo5FNoLPZiuNYuCdQ",
          "integrity_value":
            "QbYksTWNZTcMfJMLoGB_aTCA0-IuNObm19_VdpabviM"
        }
      },
      "initialization_vector":
        "AxY8DctDaGlsbGljb3RoZQ",
      "ciphertext":
        "Rxsjg6PIExcmGSF7LnSEkDqWIKfAwlwZz2XpabV5PwQsolKwEauWYZNE9QlhZJE
        Z"
    }
  ]
}

```

5. IANA Considerations

This specification makes no requests of IANA.

6. Security Considerations

The security considerations for this specification are the same as those for the JSON Web Encryption (JWE) [JWE] specification.

7. References

7.1. Normative References

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

December 2012.

- [JWE] Jones, M., Rescorla, E., and J. Hildebrand, "JSON Web Encryption (JWE)", draft-ietf-jose-json-web-encryption (work in progress), December 2012.
- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.
- [RFC4627] Crockford, D., "The application/json Media Type for JavaScript Object Notation (JSON)", RFC 4627, July 2006.

7.2. Informative References

- [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.
- [JWS-JS] Jones, M., Bradley, J., and N. Sakimura, "JSON Web Signature JSON Serialization (JWS-JS)", draft-jones-jose-jws-json-serialization (work in progress), December 2012.

Appendix A. Acknowledgements

JSON serializations for encrypted content were previously explored by JSON Simple Encryption [JSE] and JavaScript Message Security Format [I-D.rescorla-jsms].

Appendix B. Open Issues

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

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

- o Track changes that occur in the JWE spec.

Appendix C. Document History

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

-04

- o Added seriesInfo information to Internet Draft references.

-03

- o Updated values for example AES CBC calculations.

-02

- o Changed to use an array of structures for per-recipient values, rather than a set of parallel arrays.
- 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.

-01

- o Added a complete JWE-JS example.
- o Generalized language to refer to Message Authentication Codes (MACs) rather than Hash-based Message Authentication Codes (HMACs).

-00

- o Renamed draft-jones-json-web-encryption-json-serialization to draft-jones-jose-jwe-json-serialization to have "jose" be in the document name so it can be included in the Related Documents list at <http://datatracker.ietf.org/wg/jose/>. No normative changes.

draft-jones-json-web-encryption-json-serialization-02

- o Updated examples to track updated algorithm properties in the JWA spec.
- o Tracked editorial changes made to the JWE spec.

draft-jones-json-web-encryption-json-serialization-01

- o Tracked changes between JOSE JWE draft -00 and -01, which added an integrity check for non-Authenticated Encryption algorithms.

draft-jones-json-web-encryption-json-serialization-00

- o Created the initial version incorporating JOSE working group input and drawing from the JSON Serialization previously proposed in draft-jones-json-web-token-01.

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: June 30, 2013

M. Jones
Microsoft
J. Bradley
independent
N. Sakimura
Nomura Research Institute
December 27, 2012

JSON Web Signature JSON Serialization (JWS-JS)
draft-jones-jose-jws-json-serialization-04

Abstract

The JSON Web Signature JSON Serialization (JWS-JS) is a means of representing content secured with digital signatures or Message Authentication Codes (MACs) using JavaScript Object Notation (JSON) data structures. This specification describes a means of representing secured content as a JSON data object (as opposed to the JWS specification, which uses a compact serialization with a URL-safe representation). It enables multiple digital signatures and/or MACs to be applied to the same content (unlike JWS). Cryptographic algorithms and identifiers used with this specification are described in the separate JSON Web Algorithms (JWA) specification. The JSON Serialization for related encryption functionality is described in the separate JSON Web Encryption JSON Serialization (JWE-JS) 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 June 30, 2013.

Copyright Notice

Copyright (c) 2012 IETF Trust and the persons identified as the

document authors. All rights reserved.

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

Table of Contents

1. Introduction	3
1.1. Notational Conventions	3
2. Terminology	3
3. JSON Serialization	3
4. Example JWS-JS	4
5. IANA Considerations	5
6. Security Considerations	5
7. References	5
7.1. Normative References	5
7.2. Informative References	6
Appendix A. Acknowledgements	6
Appendix B. Open Issues	6
Appendix C. Document History	6
Authors' Addresses	7

1. Introduction

The JSON Web Signature JSON Serialization (JWS-JS) is a format for representing content secured with digital signatures or Message Authentication Codes (MACs) as a JavaScript Object Notation (JSON) [RFC4627] object. It enables multiple digital signatures and/or MACs to be applied to the same content (unlike JWS [JWS]). The digital signature and MAC mechanisms used are independent of the type of content being secured, allowing arbitrary content to be secured. Cryptographic algorithms and identifiers used with this specification are described in the separate JSON Web Algorithms (JWA) [JWA] specification. The JSON Serialization for related encryption functionality is described in the separate JSON Web Encryption JSON Serialization (JWE-JS) [JWE-JS] specification.

1.1. Notational Conventions

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

2. Terminology

This specification uses the same terminology as the JSON Web Signature (JWS) [JWS] specification.

3. JSON Serialization

The JSON Serialization represents secured content as a JSON object with a "recipients" member containing an array of per-recipient information and a "payload" member containing a shared Encoded JWS Payload value. Each member of the "recipients" array is a JSON object with a "header" member containing an Encoded JWS Header value and a "signature" member containing an Encoded JWS Signature value.

Unlike the compact serialization used by JWSs, content using the JSON Serialization MAY be secured with more than one digital signature and/or MAC value. Each is represented as an Encoded JWS Signature value in the "signature" member of an object in the "recipients" array. For each, there is an Encoded JWS Encoded Header value in the "header" member of the same object in the "recipients" array. This specifies the digital signature or MAC applied to the Encoded JWS Header value and the shared Encoded JWS Payload value to create the JWS Signature value. Therefore, the syntax is:

```
{ "recipients": [
  { "header": "<header 1 contents>",
    "signature": "<signature 1 contents>" },
  ...
  { "header": "<header N contents>",
    "signature": "<signature N contents>" } ],
  "payload": "<payload contents>"
}
```

The contents of the Encoded JWS Header, Encoded JWS Payload, and Encoded JWS Signature values are exactly as specified in JSON Web Signature (JWS) [JWS]. They are interpreted and validated in the same manner, with each corresponding "header" and "signature" value being created and validated together.

Each JWS Signature value is computed on the JWS Secured Input corresponding to the concatenation of the Encoded JWS Header, a period ('.') character, and the Encoded JWS Payload in the same manner described in the JWS specification. This has the desirable result that each Encoded JWS signature value in the "recipients" array is identical to the value that would be used for the same parameters in a JWS.

4. Example JWS-JS

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 Encoded JWS Payload used in this example is the same as used in the examples in Appendix A of JWS (with line breaks for display purposes only):

```
eyJpc3MiOiJqb2UiLA0KICJleHAiOjEzMDA4MTkzODAsDQogImh0dHA6Ly9leGFT
cGx1LmNvbS9pc19yb290Ijp0cnVlfQ
```

Two digital signatures are used in this example: an RSA SHA-256 signature, for which the header and signature values are the same as in Appendix A.2 of JWS, and an ECDSA P-256 SHA-256 signature, for which the header and signature values are the same as in Appendix A.3 of JWS. The two Decoded JWS Header Segments used are:

```
{ "alg": "RS256" }
```

and:

```
{ "alg": "ES256" }
```

Since the computations of the JWS Header and JWS Signature values are the same as in Appendix A.2 and Appendix A.3 of JWS, they are not repeated here.

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

```
{ "recipients": [
  { "header": "eyJhbGciOiJSUzI1NiJ9",
    "signature":
      "cC4hiUPoj9Eetdgtv3hF80EGrhuB__dzERat0XF9g2VtQgr9PJbu3XOiZj5RZ
      mh7AAuHIm4Bh-0Qc_lF5YKt_08W2Fp5jujGbds9uJdbF9CUAr7tldnZcAcQjb
      KBYNX4BAynRFdiuB--f_nZLgrnbyTyWz075vRK5h6xBArLIARNPvkSjtQBMHl
      b1L07Qe7K0GarZRmB_eSN9383LcOLn6_dO--xi12jzDwusC-eOkHWESqtfZES
      c6BfI7noOPqvhJlphCnvWh6IeYI2w9QOYEUpUTI8np6LbgGY9Fs98rqVt5AX
      LIhWkWywlvmtVrBp0igcN_IoypGLUPQGe77Rw" },
  { "header": "eyJhbGciOiJFUzI1NiJ9",
    "signature":
      "DtEhU3ljBEg8L38VWafUAqOyKAM6-Xx-F4GawxaepmXFCgfTjDxw5djxLa8IS
      lSApmWQxfKTUJqPP3-Kg6NU1Q" } ],
  "payload":
    "eyJpc3MiOiJqb2UiLA0KICJleHAiOjEzMDA4MTkzODAsDQogImh0dHA6Ly9leGF
    tcGx1LmNvbS9pc19yb290Ijp0cnVlfQ"
}
```

5. IANA Considerations

This specification makes no requests of IANA.

6. Security Considerations

The security considerations for this specification are the same as those for the JSON Web Signature (JWS) [JWS] specification.

7. References

7.1. Normative References

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

- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.
- [RFC4627] Crockford, D., "The application/json Media Type for JavaScript Object Notation (JSON)", RFC 4627, July 2006.

7.2. Informative References

- [JSS] Bradley, J. and N. Sakimura (editor), "JSON Simple Sign", September 2010.
- [JWE-JS] Jones, M., "JSON Web Encryption JSON Serialization (JWE-JS)", draft-jones-jose-jwe-json-serialization (work in progress), December 2012.
- [MagicSignatures] Panzer (editor), J., Laurie, B., and D. Balfanz, "Magic Signatures", January 2011.

Appendix A. Acknowledgements

JSON serializations for secured content were previously explored by Magic Signatures [MagicSignatures] and JSON Simple Sign [JSS].

Appendix B. Open Issues

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

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

- o Track changes that occur in the JWS spec.

Appendix C. Document History

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

-04

- o Added seriesInfo information to Internet Draft references.

-03

- o Updated references.

-02

- o Changed to use an array of structures for per-recipient values, rather than a set of parallel arrays.

-01

- o Generalized language to refer to Message Authentication Codes (MACs) rather than Hash-based Message Authentication Codes (HMACs).

-00

- o Renamed draft-jones-json-web-signature-json-serialization to draft-jones-jose-jws-json-serialization to have "jose" be in the document name so it can be included in the Related Documents list at <http://datatracker.ietf.org/wg/jose/>. No normative changes.

draft-jones-json-web-signature-json-serialization-02

- o Tracked editorial changes made to the JWS spec.

draft-jones-json-web-signature-json-serialization-01

- o Corrected the Magic Signatures reference.

draft-jones-json-web-signature-json-serialization-00

- o Created the initial version incorporating JOSE working group input and drawing from the JSON Serialization previously proposed in draft-jones-json-web-token-01.

Authors' Addresses

Michael B. Jones
Microsoft

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

John Bradley
independent

Email: ve7jtb@ve7jtb.com

Nat Sakimura
Nomura Research Institute
Email: n-sakimura@nri.co.jp

Network Working Group
Internet-Draft
Intended status: Standards Track
Expires: January 5, 2015

D. McGrew
J. Foley
Cisco Systems
K. Paterson
Royal Holloway, University of
London
July 4, 2014

Authenticated Encryption with AES-CBC and HMAC-SHA
draft-mcgrew-aead-aes-cbc-hmac-sha2-05.txt

Abstract

This document specifies algorithms for authenticated encryption with associated data (AEAD) that are based on the composition of the Advanced Encryption Standard (AES) in the Cipher Block Chaining (CBC) mode of operation for encryption, and the HMAC-SHA message authentication code (MAC).

These are randomized encryption algorithms, and thus are suitable for use with applications that cannot provide distinct nonces to each invocation of the AEAD encrypt operation.

Status of this Memo

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

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

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

This Internet-Draft will expire on January 5, 2015.

Copyright Notice

Copyright (c) 2014 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	3
1.1. History	3
1.2. Conventions Used In This Document	4
2. CBC-HMAC algorithms	5
2.1. Encryption	5
2.2. Decryption	7
2.3. Length	8
2.4. AEAD_AES_128_CBC_HMAC_SHA_256	8
2.5. AEAD_AES_192_CBC_HMAC_SHA_384	9
2.6. AEAD_AES_256_CBC_HMAC_SHA_384	9
2.7. AEAD_AES_256_CBC_HMAC_SHA_512	10
2.8. Summary	10
3. Randomness Requirements	11
4. Rationale	12
5. Test Cases	14
5.1. AEAD_AES_128_CBC_HMAC_SHA256	14
5.2. AEAD_AES_192_CBC_HMAC_SHA384	16
5.3. AEAD_AES_256_CBC_HMAC_SHA384	18
5.4. AEAD_AES_256_CBC_HMAC_SHA512	20
6. Security Considerations	22
7. Acknowledgements	24
8. References	25
8.1. Normative References	25
8.2. Informative References	25
Appendix A. CBC Encryption and Decryption	28
Appendix B. Alternative Interface for Legacy Encoding	30
Authors' Addresses	31

1. Introduction

Authenticated Encryption (AE) [BN00] is a form of encryption that, in addition to providing confidentiality for the plaintext that is encrypted, provides a way to check its integrity and authenticity. This combination of features can, when properly implemented, provide security against adversaries who have access to full decryption capabilities for ciphertexts of their choice, and access to full encryption capabilities for plaintexts of their choice. The strong form of security provided by AE is known to be robust against a large class of adversaries for general purpose applications of AE, including applications such as securing network communications over untrusted networks. The strong security properties of AE stand in contrast to the known weaknesses of "encryption only" forms of encryption, see [B96][YHR04] [DP07] for examples.

Authenticated encryption with Associated Data, or AEAD [R02], adds the ability to check the integrity and authenticity of some associated data (sometimes called "additional authenticated data") for which confidentiality is not required (or is not desirable). While many approaches to building AEAD schemes are known, a particularly simple, well-understood, and cryptographically strong method is to employ an "Encrypt-then-MAC" construction. This document defines new AEAD algorithms of this general type, using the interface defined in [RFC5116], based on the Advanced Encryption Standard (AES) [FIPS197] in the Cipher Block Chaining (CBC) mode of operation [SP800-38] and HMAC using the Secure Hash Algorithm (SHA) [FIPS186-2], with security levels of 128, 192, and 256 bits.

Comments on this version are requested and should be forwarded to the IRTF Crypto Forum Research Group (CFRG). An earlier version of this document benefited from some review from that group.

1.1. History

This subsection describes the revision history of this Internet Draft. It should be removed by the RFC Editor before publication as an RFC.

The changes of version 05 from version 04 consist only of changes in Appendix A and the test cases. A variable Q was defined to make the legacy encoding more clear, after discussion between the authors and Mike Jones.

The changes of version 02 from version 01 are:

Added test cases for each of the five operational modes.

Added John as a coauthor.

Adds a legacy-style interface in Appendix B.

The changes of version 01 from version 00 are:

MIN_LEN_A and associated logic was eliminated.

Padding String (PS) typo corrected in Section 2.1.

Decryption Step 3 refers to the appropriate step in the encryption process.

Random IV min-entropy clarified in Section 3.

HMAC keys are now the same size as the truncated output (128 or 256 bits). Previously, the HMAC keys were the same size as the full hash output (256, 384, or 512 bits).

An algorithm based on the combination of AES-256 and HMAC-SHA-384 has been added, for compatibility with draft-burgin-kerberos-aes-cbc-hmac-sha2.

The test cases in the previous version are no longer valid, and thus have been removed. New test cases have been computed (and the authors thank John Foley for this contribution) but have not been included, pending confirmation from a second, independent implementation.

1.2. Conventions Used In This Document

We use the following notational conventions.

$\text{CBC-ENC}(X,P)$ denotes the CBC encryption of P using the cipher with the key X

$\text{MAC}(Y, M)$ denotes the application of the Message Authentication Code (MAC) to the message M , using the key Y

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

$\text{len}(X)$ denotes the number of bits in the string X , expressed as an unsigned integer in network byte order.

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 [RFC2119].

2. CBC-HMAC algorithms

This section defines CBC-HMAC, an algorithm based on the the encrypt-then-MAC method defined in Section 4.3 of [BN00]. That method constructs a randomized AEAD algorithm out of a randomized cipher, such as a block cipher mode of operation that uses a random initialization vector, and a MAC.

Section 2.1 and Section 2.2 define the CBC-HMAC encryption and decryption algorithms, without specifying the particular block cipher or hash function to be used. Section 2.4, Section 2.5, and Section 2.7 define instances of CBC-HMAC that specify those details.

2.1. Encryption

We briefly recall the encryption interface defined in Section 2 of [RFC5116]. The AEAD encryption algorithm takes as input four octet strings: a secret key K, a plaintext P, associated data A, and a nonce N. An authenticated ciphertext value is provided as output. The data in the plaintext are encrypted and authenticated, and the associated data are authenticated, but not encrypted. The key **MUST** be generated in a way that is uniformly random or pseudorandom; guidance on random sources is provided in [RFC4086].

In CBC-HMAC, the nonce N **MUST** be a zero-length string; a nonce is not needed and is not used (see Section 4 for further background).

The CBC-HMAC 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 2.4 and Section 2.5). The number of octets in the input key K is the sum of MAC_KEY_LEN and ENC_KEY_LEN. When generating the secondary keys from K, MAC_KEY and ENC_KEY **MUST NOT** overlap.

2. Prior to CBC encryption, the plaintext P is padded by appending a padding string PS to that data, to ensure that $\text{len}(P \parallel PS)$ is a multiple of 128, as is needed for the CBC operation. The value of PS is as follows:

$PS = 01$	if $\text{len}(P) \bmod 128 = 120,$
$PS = 0202$	if $\text{len}(P) \bmod 128 = 112,$
$PS = 030303$	if $\text{len}(P) \bmod 128 = 104,$
$PS = 04040404$	if $\text{len}(P) \bmod 128 = 96,$
$PS = 0505050505$	if $\text{len}(P) \bmod 128 = 88,$
$PS = 060606060606$	if $\text{len}(P) \bmod 128 = 80,$
$PS = 07070707070707$	if $\text{len}(P) \bmod 128 = 72,$
$PS = 0808080808080808$	if $\text{len}(P) \bmod 128 = 64,$
$PS = 090909090909090909$	if $\text{len}(P) \bmod 128 = 56,$
$PS = 0A0A0A0A0A0A0A0A0A$	if $\text{len}(P) \bmod 128 = 48,$
$PS = 0B0B0B0B0B0B0B0B0B0B$	if $\text{len}(P) \bmod 128 = 40,$
$PS = 0C0C0C0C0C0C0C0C0C0C$	if $\text{len}(P) \bmod 128 = 32,$
$PS = 0D0D0D0D0D0D0D0D0D0D0D$	if $\text{len}(P) \bmod 128 = 24,$
$PS = 0E0E0E0E0E0E0E0E0E0E0E$	if $\text{len}(P) \bmod 128 = 16,$
$PS = 0F0F0F0F0F0F0F0F0F0F0F$	if $\text{len}(P) \bmod 128 = 8,$
$PS = 1010101010101010101010101010$	if $\text{len}(P) \bmod 128 = 0.$

Note that padding **MUST** be added to the plaintext; if the number of bits in P is a multiple of 128, then 128 bits of padding will be added.

3. The plaintext and appended padding $P \parallel PS$ is CBC encrypted using ENC_KEY as the key, as described in Appendix A. We denote the ciphertext output from this step as S .
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 ciphertext S 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 T .

6. The AEAD Ciphertext consists of the string S , with the string T appended to it. This Ciphertext is returned as the output of the AEAD encryption operation.

The encryption process can be illustrated as follows. Here P , A , and C denote the AEAD plaintext, associated data, and ciphertext, respectively.

$MAC_KEY = \text{initial } MAC_KEY_LEN \text{ bytes of } K$

$ENC_KEY = \text{final } ENC_KEY_LEN \text{ bytes of } K$

$S = \text{CBC-ENC}(ENC_KEY, P \parallel PS),$

$T = \text{MAC}(MAC_KEY, A \parallel S \parallel AL),$

$C = S \parallel T.$

2.2. Decryption

The authenticated decryption operation has four inputs: K , N , and A , as defined above, and the Ciphertext C . As discussed above, N is an empty string in AES-CBC and is not used below. 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 takes 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 2.1.
2. The final T_LEN octets are stripped from C . Here T_LEN denotes the number of octets in the MAC, which is a fixed parameter of the AEAD algorithm. We denote the initial octets of C as S , and denote the final T_LEN octets as T .
3. The integrity and authenticity of A and C are checked by computing HMAC with the inputs as in Step 6 of Section 2.1. The value T , from the previous step, is compared to the HMAC output, using a comparison routine that takes constant time to execute. If those values are identical, then A and C are considered valid, and the processing continues. 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.
4. The value S is CBC decrypted, as described in Appendix A, using the value ENC_KEY as the decryption key.
5. The padding string is stripped from the resulting plaintext. Note that the length of PS can be inferred from the value of the final octet of $P \parallel PS$, if that value is between 01 and 10

(hexadecimal). If the final octet has a value outside that range, then all of the data used in the processing of the message is zeroized and discarded, and the AEAD decryption operation returns an indication that it failed, and the operation halts.

6. The plaintext value is returned.

2.3. Length

The length of the ciphertext can be inferred from that of the plaintext. The number L of octets in the ciphertext is given by

$$L = 16 * (\text{floor}(M / 16) + 2)$$

where M denotes the number of octets in the plaintext, and the function `floor()` rounds its argument down to the nearest integer. This fact is useful to applications that need to reserve space for a ciphertext within a message or data structure.

2.4. AEAD_AES_128_CBC_HMAC_SHA_256

This algorithm is randomized; each invocation of the encrypt operation makes use of a random value (the IV described in Appendix A). It is based on the CBC-HMAC algorithm detailed above, and uses the HMAC message authentication code [RFC2104] with the SHA-256 hash function [FIPS186-2] 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 the Advanced Encryption Standard (AES) [FIPS197] block cipher in CBC mode.

The input key K is 32 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. Test cases for HMAC-SHA-256 are provided in [RFC4231].

The lengths of the inputs are restricted as follows:

`K_LEN` is 32 octets,

`P_MAX` is $2^{64} - 1$ octets,

`A_MAX` is $2^{64} - 1$ octets,

N_MIN and N_MAX are zero octets,

C_MAX is $2^{64} + 47$ octets.

2.5. AEAD_AES_192_CBC_HMAC_SHA_384

AEAD_AES_192_CBC_HMAC_SHA_384 is based on AEAD_AES_128_CBC_HMAC_SHA_256, but with the following differences:

AES-192 is used instead of AES-128.

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

ENC_KEY_LEN is 24 octets.

MAC_KEY_LEN is 24 octets.

The length of the input key K is 48 octets.

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

The input length restrictions are as for AEAD_AES_CBC_128_HMAC_SHA_256.

2.6. AEAD_AES_256_CBC_HMAC_SHA_384

AEAD_AES_256_CBC_HMAC_SHA_384 is based on AEAD_AES_128_CBC_HMAC_SHA_256, but with the following differences:

AES-256 is used instead of AES-128.

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

ENC_KEY_LEN is 32 octets.

MAC_KEY_LEN is 24 octets.

The length of the input key K is 56 octets.

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

The input length restrictions are as for AEAD_AES_CBC_128_HMAC_SHA_256.

2.7. AEAD_AES_256_CBC_HMAC_SHA_512

AEAD_AES_256_CBC_HMAC_SHA_512 is based on AEAD_AES_128_CBC_HMAC_SHA_256, but with the following differences:

AES-256 is used instead of AES-128.

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

ENC_KEY_LEN is 32 octets.

MAC_KEY_LEN is 32 octets.

The length of the input key K is 64 octets.

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

The input length restrictions are as for AEAD_AES_CBC_128_HMAC_SHA_256.

2.8. Summary

The parameters of the CBC-HMAC algorithms are summarized in the following table.

algorithm	ENC_KEY_LEN	MAC_KEY_LEN	T_LEN
AEAD_AES_128_CBC_HMAC_SHA_256	16	16	16
AEAD_AES_192_CBC_HMAC_SHA_384	24	24	24
AEAD_AES_256_CBC_HMAC_SHA_384	32	24	24
AEAD_AES_256_CBC_HMAC_SHA_512	32	32	32

3. Randomness Requirements

Each IV MUST be unpredictable to the adversary. It MAY be chosen uniformly at random, in which case it SHOULD have min-entropy within one bit of `len(IV)`. Alternatively, it MAY be generated pseudorandomly, using any method that provides the same level of security as the block cipher in use. However, if a pseudorandom method is used, that method MUST NOT make use of `ENC_KEY` or `MAC_KEY`.

4. Rationale

The CBC-HMAC AEAD algorithms defined in this note are intended to be useful in the following applications:

systems that have the CBC and HMAC algorithms available, but do not have dedicated AEAD algorithms such as GCM or CCM [RFC5116],

scenarios in which AEAD is useful, but it is undesirable to have the application maintain a deterministic nonce; see Section 4 of [RFC5116] for more background,

new systems, such as JSON Cryptography and W3C Web Crypto, which can omit unauthenticated symmetric encryption altogether by providing CBC and HMAC through an AEAD interface.

These algorithms are not intended to replace existing uses of AES-CBC and HMAC, except in those circumstances where the existing use is not sufficiently secure or sufficiently general-purpose.

The algorithms in this note truncate the HMAC output to half of the size of the output of the underlying hash function. This size is the recommended minimum (see Section 5 of [RFC2104]), and this parameter choice has withstood the test of time.

The length of the associated data input A is included in the HMAC input to ensure that the encrypter and the decrypter have the same understanding of that length. Because of this, an attacker cannot trick the receiver into interpreting the initial bytes of C as the final bytes of A, or vice-versa.

The padding method used in this note is based on that of Privacy Enhanced Mail (PEM) and the Public Key Cryptography Standards (PKCS), because it is implemented in many environments.

The encrypt-then-MAC method is used because of its better security properties. It would be possible to define AEAD algorithms based on the MAC-encode-encrypt (MEE) method that is used by the Transport Layer Security (TLS) protocol [RFC5246]. That alternative would provide more code-sharing opportunities for an implementation that is co-resident with a TLS implementation. It is possible (but tricky) to implement MEE in a way that provides good security, as was shown in [PRS11]. But its negatives outweigh its positives; its security is inadequate for some parameter choices, and it has proven to be very difficult to implement in a way that resists padding oracle and related timing attacks [V02] [CHVV03] [M04] [DP10] [AP12]. For future uses of CBC and HMAC, it is better to use encrypt-then-MAC.

This note uses HMAC-SHA-2 because it is widely deployed, it is mandated in newer standards, and because SHA1 is being deprecated. It has been recently announced that the SHA-3 standard will be based on KECCAK, but this note does not incorporate that hash function. To do so would be to speculate on the final form of the SHA-3 standard. In addition, while the use of KECCAK as a hash function is straightforward, there are multiple options for its use in authenticated encryption. The focus of this note is the definition of AEAD algorithms based on currently used cryptographic mechanisms, so SHA-3 is out of scope.

5. Test Cases

5.1. AEAD_AES_128_CBC_HMAC_SHA256

```
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

PS =      10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10

AL =      00 00 00 00 00 00 01 50
```


Q = 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

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

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

C = 1a f3 8c 2d c2 b9 6f fd d8 66 94 09 23 41 bc 04
 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
 65 2c 3f a3 6b 0a 7c 5b 32 19 fa b3 a3 0b c1 c4

5.2. AEAD_AES_192_CBC_HMAC_SHA384

```

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

PS =      10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10

AL =      00 00 00 00 00 00 01 50

```

Q = 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

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

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 = 1a f3 8c 2d c2 b9 6f fd d8 66 94 09 23 41 bc 04
 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
 84 90 ac 0e 58 94 9b fe 51 87 5d 73 3f 93 ac 20
 75 16 80 39 cc c7 33 d7

5.3. AEAD_AES_256_CBC_HMAC_SHA384

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

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 30 31 32 33 34 35 36 37

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

PS = 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10

AL = 00 00 00 00 00 00 01 50

Q = 89 31 29 b0 f4 ee 9e b1 8d 75 ed a6 f2 aa a9 f3
60 7c 98 c4 ba 04 44 d3 41 62 17 0d 89 61 88 4e
58 f2 7d 4a 35 a5 e3 e3 23 4a a9 94 04 f3 27 f5
c2 d7 8e 98 6e 57 49 85 8b 88 bc dd c2 ba 05 21
8f 19 51 12 d6 ad 48 fa 3b 1e 89 aa 7f 20 d5 96
68 2f 10 b3 64 8d 3b b0 c9 83 c3 18 5f 59 e3 6d
28 f6 47 c1 c1 39 88 de 8e a0 d8 21 19 8c 15 09
77 e2 8c a7 68 08 0b c7 8c 35 fa ed 69 d8 c0 b7
d9 f5 06 23 21 98 a4 89 a1 a6 ae 03 a3 19 fb 30

S = 1a f3 8c 2d c2 b9 6f fd d8 66 94 09 23 41 bc 04
89 31 29 b0 f4 ee 9e b1 8d 75 ed a6 f2 aa a9 f3
60 7c 98 c4 ba 04 44 d3 41 62 17 0d 89 61 88 4e
58 f2 7d 4a 35 a5 e3 e3 23 4a a9 94 04 f3 27 f5
c2 d7 8e 98 6e 57 49 85 8b 88 bc dd c2 ba 05 21
8f 19 51 12 d6 ad 48 fa 3b 1e 89 aa 7f 20 d5 96
68 2f 10 b3 64 8d 3b b0 c9 83 c3 18 5f 59 e3 6d
28 f6 47 c1 c1 39 88 de 8e a0 d8 21 19 8c 15 09
77 e2 8c a7 68 08 0b c7 8c 35 fa ed 69 d8 c0 b7
d9 f5 06 23 21 98 a4 89 a1 a6 ae 03 a3 19 fb 30

T = dd 13 1d 05 ab 34 67 dd 05 6f 8e 88 2b ad 70 63
7f 1e 9a 54 1d 9c 23 e7

C = 1a f3 8c 2d c2 b9 6f fd d8 66 94 09 23 41 bc 04
89 31 29 b0 f4 ee 9e b1 8d 75 ed a6 f2 aa a9 f3
60 7c 98 c4 ba 04 44 d3 41 62 17 0d 89 61 88 4e
58 f2 7d 4a 35 a5 e3 e3 23 4a a9 94 04 f3 27 f5
c2 d7 8e 98 6e 57 49 85 8b 88 bc dd c2 ba 05 21
8f 19 51 12 d6 ad 48 fa 3b 1e 89 aa 7f 20 d5 96
68 2f 10 b3 64 8d 3b b0 c9 83 c3 18 5f 59 e3 6d
28 f6 47 c1 c1 39 88 de 8e a0 d8 21 19 8c 15 09
77 e2 8c a7 68 08 0b c7 8c 35 fa ed 69 d8 c0 b7
d9 f5 06 23 21 98 a4 89 a1 a6 ae 03 a3 19 fb 30
dd 13 1d 05 ab 34 67 dd 05 6f 8e 88 2b ad 70 63
7f 1e 9a 54 1d 9c 23 e7

5.4. AEAD_AES_256_CBC_HMAC_SHA512

```
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

PS =      10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10

AL =      00 00 00 00 00 00 01 50
```

Q = 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

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

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

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

6. Security Considerations

The algorithms defined in this document use the generic composition of CBC encryption with HMAC authentication, with the encrypt-then-MAC method defined in Section 4.3 of [BN00]. This method has sound and well-understood security properties; for details, please see that reference. Note that HMAC is a good pseudorandom function and is "strongly unforgeable", and thus meets all of the security goals of that reference.

Implementations of the encryption and decryption algorithms should avoid side channels that would leak information about the secret key. To avoid timing channels, the processing time should be independent of the secret key. The Encrypt-then-MAC construction used in this note has some inherent strength against timing attacks because, during the decryption operation, the authentication check is computed before the plaintext padding is processed. However, the security of the algorithm still relies on the absence of timing channels in both CBC and HMAC. Additionally, comparison between the authentication tag T and the HMAC output should be done using a constant-time operation.

During the decryption process, the inputs A and C are mapped into the input of the HMAC algorithm. It is essential for security that each possible input to the MAC algorithm corresponds unambiguously to exactly one pair (A, C) of possible inputs. The fact that this property holds can be verified as follows. The HMAC input is $X = A || C || \text{len}(A)$. Let (A, C) and (A', C') denote two distinct input pairs, in which either 1) $A \neq A'$ and $C = C'$, 2) $C \neq C'$ and $A = A'$, or 3) both inequalities hold. We also let $X' = A' || C' || \text{len}(A')$. In cases 1 and 2, $X \neq X'$ follows immediately. In case 3, if $\text{len}(A) \neq \text{len}(A')$, then $X \neq X'$ directly. If $\text{len}(A) = \text{len}(A')$, then $X \neq X'$ follows from the fact that the initial $\text{len}(A)$ bits of X and X' must be distinct.

There are security benefits to providing both confidentiality and authentication in a single atomic operation, as done in this note. This tight binding prevents subtle attacks such as the padding oracle attack.

As with any block cipher mode of operation, the security of AES-CBC degrades as the amount of data that is process increases. Each fixed key value SHOULD NOT be used to protect more than 2^{64} bytes of data. This limit ensures that the AES-CBC algorithm will stay under the birthday bound, i.e. because of the limit, it is unlikely that there will be two AES plaintext inputs that are equal. (If this event occurs, information about the colliding plaintexts is leaked, so it is desirable to bound the amount of plaintext processed in order to

make it unlikely.)

7. Acknowledgements

Thanks are due to Matt Miller for his constructive feedback, Kelly Burgin, Michael Peck, and Mike Jones for their suggestions and help, and Jim Schaad, Rob Napier, James Manger, and David Jacobson for their excellent review and suggestions.

8. References

8.1. Normative References

- [FIPS186-2] "FIPS 180-2: Secure Hash Standard," Federal Information Processing Standard
(FIPS) <http://www.itl.nist.gov/fipspubs/fip180-1.htm>.
- [FIPS197] "FIPS 197: Advanced Encryption Standard (AES)", Federal Information Processing Standard
(FIPS) <http://www.itl.nist.gov/fipspubs/fip197.htm>.
- [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.
- [RFC4231] Nystrom, M., "Identifiers and Test Vectors for HMAC-SHA-224, HMAC-SHA-256, HMAC-SHA-384, and HMAC-SHA-512", RFC 4231, December 2005.
- [RFC4868] Kelly, S. and S. Frankel, "Using HMAC-SHA-256, HMAC-SHA-384, and HMAC-SHA-512 with IPsec", RFC 4868, May 2007.
- [RFC5116] McGrew, D., "An Interface and Algorithms for Authenticated Encryption", RFC 5116, January 2008.

8.2. Informative References

- [AP12] Paterson, K. and N. AlFardan, "Plaintext-Recovery Attacks Against Datagram TLS", Network and Distributed System Security Symposium (NDSS) 2012 <http://www.isg.rhul.ac.uk/~kp/dtls.pdf>, 2012.
- [B96] Bellare, S., "Problem areas for the IP security protocols", Proceedings of the Sixth Usenix Unix Security Symposium <https://www.cs.columbia.edu/~smb/papers/badesp.pdf>, 1996.
- [BN00] "Authenticated encryption: Relations among notions and analysis of the generic composition paradigm", Proceedings of ASIACRYPT 2000, Springer-Verlag, LNCS 1976, pp. 531-545 <http://www-cse.ucsd.edu/users/mihir/papers/oem.html>.
- [CHVV03] Vaudenay, S., Canvel, B., Hiltgen, A., and M. Vuagnoux,

- "Password Interception in a SSL/TLS Channel", CRYPTO 2003 <http://lasecwww.epfl.ch/pub/lasec/doc/CHVV03.ps>, 2003.
- [DP07] Paterson, K. and J. Degabriele, "Attacking the IPsec Standards in Encryption-only Configurations", IEEE Symposium on Privacy and Security <http://eprint.iacr.org/2007/125.pdf>, 2007.
- [DP10] Paterson, K. and J. Degabriele, "On the (in)security of IPsec in MAC-then-encrypt configurations.", ACM Conference on Computer and Communications Security (ACM CCS) 2010 <http://www.isg.rhul.ac.uk/~kp/CCSIPsecfinal.pdf>, 2010.
- [M04] Moeller, B., "Security of CBC Ciphersuites in SSL/TLS: Problems and Countermeasures", Web Page <http://www.openssl.org/~bodo/tls-cbc.txt>, 2004.
- [PRS11] Paterson, K., Ristenpart, T., and T. Shrimpton, "Tag Size Does Matter: Attacks and Proofs for the TLS Record Protocol", IEEE Symposium on Security and Privacy 2012 <http://www.isg.rhul.ac.uk/~kp/mee-comp.pdf>, January 2012.
- [R02] "Authenticated encryption with Associated-Data", Proceedings of the 2002 ACM Conference on Computer and Communication Security (CCS'02), pp. 98-107, ACM Press, 2002. <http://www.cs.ucdavis.edu/~rogaway/papers/ad.pdf>.
- [RFC4086] Eastlake, D., Schiller, J., and S. Crocker, "Randomness Requirements for Security", BCP 106, RFC 4086, June 2005.
- [RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, August 2008.
- [SP800-38] Dworkin, M., "NIST Special Publication 800-38: Recommendation for Block Cipher Modes of Operation", U.S. National Institute of Standards and Technology <http://csrc.nist.gov/publications/nistpubs/800-38a/sp800-38a.pdf>.
- [V02] Vaudenay, S., "Security Flaws Induced by CBC Padding - Applications to SSL, IPSEC, WTLS", EUROCRYPT 2002 http://lasecwww.epfl.ch/php_code/publications/search.php?ref=Vau02a, 2002.
- [YHR04] Yu, T., Hartman, S., and K. Raeburn, "The Perils of

Unauthenticated Encryption: Kerberos Version 4", Network and Distributed Security Symposium (NDSS) 2004 <http://web.mit.edu/tlyu/papers/krb4peril-ndss04.pdf>, 2004.

Appendix A. CBC Encryption and Decryption

The Cipher Block Chaining (CBC) mode of operation is defined in Section 6.2 of [SP800-38]. This section recalls how that mode works, for the convenience of the reader. The following notation is used:

K denotes the key of the underlying block cipher,

The function $CIPHER(K, P)$ denotes the encryption of the block P with the block cipher, where P contains exactly b bits,

The function $CIPHER-INV(K, Q)$ denotes the decryption of the block Q with the block cipher, where Q contains exactly b bits; this is the inverse operation of $CIPHER()$, and $CIPHER-INV(K, CIPHER(K, P)) = P$ for all P and all K ,

P_1, P_2, \dots, P_n denotes the sequence of plaintext blocks, where each block contains exactly b bits,

Q_1, Q_2, \dots, Q_n denotes the sequence of ciphertext blocks, where each block contains exactly b bits,

P_i and Q_i denote the i th blocks of the plaintext, and

IV denotes the initialization vector, which contains exactly b bits.

The CBC encryption operation (denoted as CBC-ENC) takes as input a sequence of n plaintext blocks and produces a sequence of $n + 1$ ciphertext blocks as follows:

$$\begin{aligned} IV &= \text{random} \\ Q_i &= \begin{cases} CIPHER(K, P_i \text{ XOR } IV) & \text{if } i=0, \\ CIPHER(K, P_i \text{ XOR } Q_{i-1}) & \text{if } i=1, 2, \dots, n. \end{cases} \end{aligned}$$

The operation $CBC-ENC(K, P_1 || P_2 || \dots || P_n)$ returns the value $IV || Q_1 || Q_2 || \dots || Q_n$. Note that the returned value is one block longer than the input value.

The IV MUST be generated using a uniformly random process, or a pseudorandom process with a cryptographic strength equivalent to that of the underlying block cipher; see [RFC4086] for background on random sources. It MUST NOT be predictable to an attacker; in particular, it MUST NOT be set to the value of any previous ciphertext blocks.

The CBC decryption operation (denoted as CBC-DEC) takes as input an octet string whose length is a multiple of b bits, decomposes it as

IV || Q₁ || Q₂ || ... || Q_m, then produces a sequence of m plaintext blocks as follows:

$$P_i = \begin{cases} \text{CIPHER-INV}(K, Q_i) \text{ XOR IV} & \text{if } i=1. \\ \text{CIPHER-INV}(K, Q_i) \text{ XOR } Q_{i-1} & \text{if } i=2, \dots, m. \end{cases}$$

The operation CBC-DEC(K, IV || Q₁ || Q₂ || ... || Q_m) returns the value P₁ || P₂ || ... || P_m.

Appendix B. Alternative Interface for Legacy Encoding

In some scenarios, cryptographic data such as the ciphertext, initialization vector, and message authentication tag are encoded separately. To allow for the use of the algorithms defined in this document in such scenarios, this appendix describes an interface in which those data elements are discrete. New implementations SHOULD NOT use this interface, because it is incompatible with other authenticated encryption methods and is more complex; however, it MAY be useful in scenarios in which the separate encoding is already in use.

The alternative interface is as follows. The inputs to the encryption operation the same as those defined in Section 2.1 (the secret key *K*, the plaintext *P*, the associated data *A*). The outputs of the encryption operation are:

- the initialization vector *IV* as defined in Appendix A,
- the ciphertext *C*, as defined in Appendix A, and
- the message authentication tag *T*, as defined in Section 2.1.

The inputs to the decryption operation (in addition to *K* and *A*) are:

- the initialization vector *IV* as defined in Appendix A,
- the ciphertext *C* as defined in Appendix A, excluding the initial block *C₀* (which is equal to the *IV*), and
- the message authentication tag *T*, as defined in Section 2.1.

The output of the decryption operation are the same as that defined in Section 2.2 (either a plaintext value *P* or a special symbol *FAIL* that indicates that the inputs are not authentic).

All processing other than the encoding and decoding of *IV*, *C*, and *T* is done as defined above. In particular, the *IV* is an output of the encryption operation, rather than an input.

Authors' Addresses

David McGrew
Cisco Systems
13600 Dulles Technology Drive
Herndon, VA 20171
US

Email: mcgrew@cisco.com
URI: <http://www.mindspring.com/~dmcgrew/dam.htm>

John Foley
Cisco Systems
7025-2 Kit Creek Road
Research Triangle Park, NC 14987
US

Email: foleyj@cisco.com

Kenny Paterson
Royal Holloway, University of London
TW20 0EX
Egham, Surrey TW20 0EX
UK

Phone: +44 1784 414393
Email: Kenny.Paterson@rhul.ac.uk
URI: <http://www.isg.rhul.ac.uk/~kp/>

XMPP
Internet-Draft
Intended status: Standards Track
Expires: January 5, 2015

M. Miller
Cisco Systems, Inc.
C. Wallace
Red Hound Software, Inc.
July 4, 2014

End-to-End Object Encryption and Signatures for the Extensible Messaging
and Presence Protocol (XMPP)
draft-miller-xmpp-e2e-07

Abstract

This document defines two methods for securing objects (often referred to as stanzas) for the Extensible Messaging and Presence Protocol (XMPP), which allows for efficient asynchronous communication between two entities, each with might have multiple devices operating simultaneously. One is a method to encrypt stanzas to provide confidentiality protection; another is a method to sign stanzas to provide authentication and integrity protection. This document also defines a related protocol for entities to request the ephemeral session keys in use.

Status of This Memo

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

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

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

This Internet-Draft will expire on January 5, 2015.

Copyright Notice

Copyright (c) 2014 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	3
2. Terminology	4
3. Changes to existing clients	4
3.1. End-point procedures	4
3.2. End-point state	5
4. Key distribution	6
5. Key table	8
6. Encryption	10
6.1. Determining Support	10
6.2. Encrypting XMPP Stanzas	10
6.2.1. Prerequisites	10
6.2.2. Process	11
6.3. Decrypting XMPP Stanzas	13
6.3.1. Protocol Not Understood	13
6.3.2. Process	13
6.3.3. Insufficient Information	15
6.3.4. Failed Decryption	15
6.3.5. Timestamp Not Acceptable	16
6.3.6. Successful Decryption	17
6.4. Example - Securing a Message	17
7. Signatures	21
7.1. Determining Support	21
7.2. Signing XMPP Stanzas	22
7.2.1. Process	22
7.3. Verifying Signed XMPP Stanzas	24
7.3.1. Protocol Not Understood	24
7.3.2. Process	24
7.3.3. Insufficient Information	25
7.3.4. Failed Verification	26
7.3.5. Timestamp Not Acceptable	26
7.3.6. Successful Verification	27
7.4. Example - Signing a Message	27
8. Requesting Session Keys	30
8.1. Request Process	30
8.2. Accept Process	31
8.3. Error Conditions	33
8.4. Example of Successful Key Request	34
9. Multiple Operations	38
10. Inclusion and Checking of Timestamps	38

11. Interaction with Stanza Semantics	39
12. Interaction with Offline Storage	40
13. Mandatory-to-Implement Cryptographic Algorithms	40
14. Security Considerations	40
14.1. Storage of Encrypted Stanzas	40
14.2. Re-use of Session Master Keys	40
15. IANA Considerations	41
15.1. XML Namespaces Name for e2e Data in XMPP	41
16. References	42
16.1. Normative References	42
16.2. Informative References	43
Appendix A. Schema for urn:ietf:params:xml:ns:xmpp-e2e:6	43
Appendix B. Acknowledgements	46
Authors' Addresses	46

1. Introduction

End-to-end protection and authentication of traffic sent over the Extensible Messaging and Presence Protocol [RFC6120] is a desirable goal. Requirements and a threat analysis for XMPP encryption are provided in [E2E-REQ]. Many possible approaches to meet those (or similar) requirements have been proposed over the years, including methods based on PGP, S/MIME, SIGMA, and TLS.

Most proposals have not been able to support multiple end-points for a given recipient. As more devices support XMPP, it becomes more desirable to allow an entity to communicate with another in a more secure manner, regardless of the number of agents the entity is employing. This document specifies an approach for encrypting and signing communications between two entities which each might have multiple end-points.

A primary challenge with supporting multiple end-points is key distribution. This is complicated by the fact that some end points for a given recipient may share keys, some may use different keys, some may have no keys and some may not support encryption or signature verification at all. To address these differences, this specification defines a symmetric key table that is managed via three mechanisms that enable a key to be pushed to an end point, to be pulled from an originator or negotiated. The key table contains named master keys along with meta data describing usage of the key. Encrypted XMPP messages use a named master key to encrypt a content encryption key. Prior to decrypting a message, recipients of an encrypted message will either find the named key present in their key table (as the result of an earlier operation) or obtain the key from the sender.

Comments are solicited and should be addressed to XMPP mailing list. Information about the XMPP mailing list can be found here: <https://www.ietf.org/mailman/listinfo/xmpp>.

2. Terminology

This document inherits XMPP-related terminology from [RFC6120], JSON Web Algorithms (JWA)-related terminology from [JOSE-JWA], JSON Web Encryption (JWE)-related terminology from [JOSE-JWE], and JSON Web Key (JWK)-related terminology from [JOSE-JWK]. Security-related terms are to be understood in the sense defined in [RFC4949].

The capitalized 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 [RFC2119].

3. Changes to existing clients

3.1. End-point procedures

Existing XMPP clients will need to implement some new procedures in order to support end-to-end encryption and authentication. Changes for sending clients include:

- o Generating session master keys (SMKs)
- o Storing SMKs for use during active sessions
- o Storing SMKs to provide to peers and to support reading of saved messages (may require use of storage key)
- o Accepting requests for SMKs
- o Releasing SMKs to authorized requestors (where requests may be received from multiple different resources associated with a single peer with each resource using a different means to authenticate)
- o Generating content encryption keys (CEK)
- o Using SMK and CEK values to encrypt XMPP stanzas
- o Generating a signing key (optional)
- o Using a signing key to sign XMPP stanzas
- o Generating and using a long term storage key (optional)

Changes for receiving clients include:

- o Sending requests for SMKs to peers
- o Accepting public key to use when encrypting an SMK from peers
- o Storing SMKs for use when decrypting XMPP stanzas during active session
- o Using an SMK to decrypt a CEK used to decrypt XMPP stanzas
- o Storing SMKs retrieved from peers to support reading of saved messages (may require use of storage key)
- o Providing indication to users when encryption is in use
- o Retrieving keys required to verify signatures on signed XMPP stanzas
- o Verifying signatures and displaying indication of success/failure to user
- o Storing keys required to verify signature to support reading of saved messages (may require use of storage key)
- o Generating and using a long term storage key (optional)

3.2. End-point state

End points utilizing end-to-end encryption and signatures are required to maintain some new state information, and may find some additional information helpful to maintain. New state information includes:

- o Session master key table (required)
- o Public/private key store (required)
- o Trust anchor store (optional)
- o Intermediate certification authority (CA) store (optional)
- o Long-term storage key (optional)

Session master keys (SMKs) are used to encrypt XMPP stanzas. An end-point may have many active SMKs at any given point in time, but only one SMK active per bare JID (TODO: or should this be per full JID?). Each SMK has a name generated by the entity who generated the key.

The name MUST be unique from the generator's perspective (i.e., full JID + SMK name MUST uniquely identify a specific SMK). When a new SMK is received, any previous SMK stored for the full JID of the entity providing the SMK may be destroyed. Alternatively, previous SMKs may be preserved to support future decryption of stored messages. This specification places no requirements on handling of stored messages. Clients may re-encrypt messages under a long-term storage key, store messages as-is encrypted using an SMK or store plaintext messages.

Each end-point must have at least one public/private key pair used for SMK distribution.

A trust anchor store or intermediate CA store may be useful to support automated release of encrypted SMKs or to verify signed XMPP stanzas.

A long-term storage key may be used to either encrypt data stored in the key table or to re-encrypt encrypted messages prior to storing the message for future review.

4. Key distribution

Several different types of keys are used to support end-to-end encryption and signatures. These keys may be distinct from any keys used to authenticate to XMPP servers and include the following:

- o Session master key (SMK)
- o Content encryption keys (CEKs) for XMPP stanzas
- o Public/private key pair for SMK distribution
- o Content encryption keys for SMK distribution
- o Public/private key pair for signature generation
- o Trust anchor and intermediate certification authority (CA) public keys
- o Long-term storage key

SMKs are symmetric keys generated by an end-point prior to utilizing end-to-end encryption (see Section 6.2.1). SMKs are used to encrypt the CEK used to encrypt an XMPP stanza. SMKs are stored in the SMK table and may be distributed using one of the following mechanisms:

- o Manually pre-placed at some point prior to using end-to-end encryption
- o Released to an end-point upon request after receiving an encrypted XMPP stanza
- o Provided to an end-point using an IQ stanza sent prior to sending encrypted XMPP stanzas

CEKs for XMPP stanzas are symmetric keys generated by an end-point to encrypt an XMPP stanza (see item 5 in Section 6.2.2). CEKs are encrypted using the SMK and included with encrypted XMPP data.

Public/private key pairs for SMK distribution are asymmetric keys that may be generated by an end point, imported into an end point or used via a hardware cryptographic module. The public key is distributed to XMPP peers for use when distributing SMKs (see step 1 in Section 8.1). The public key is formatted as a JWK, which may include an X.509 certificate. An end-point MUST establish trust in a public key prior to releasing an SMK value. Trust establishment mechanisms include checking a key thumbprint provided via a trusted channel or by validating an X.509 certificate to a trust anchor. The public keys may be distributed using one of the following mechanisms:

- o Manually pre-placed prior to using for SMK release (details for manual pre-placement are not defined by this specification)
- o Presented when requesting an SMK from a peer after receiving an encrypted XMPP stanza from the peer (the peer may store the public key for use in providing future encrypted SMK values prior to using the SMK to encrypt XMPP stanzas see Section 8.1)
- o Provided upon request in response to an IQ get request in preparation for receiving encrypted XMPP stanzas (TODO: define IQ for pushing SMK)

CEKs for SMK distribution are symmetric keys generated by an end-point to encrypt an SMK (see item 3 in Section 8.2). CEKs are encrypted using the public key used for SMK distribution and included with encrypted SMK data.

Public/private key pairs for SMK distribution are asymmetric keys that may be generated by an end point, imported into an end point or used via a hardware cryptographic module (see bullet 4 of section 5.1 in [JOSE-JWE]). The public key is distributed to XMPP peers for use when verifying signatures. Trust establishment may be performed by checking a key thumbprint provided via a trusted channel or by validating an X.509 certificate to a trust anchor.

Trust anchor and intermediate CA public keys may be used to validate X.509 certificates in support of SMK release or verification of signatures on signed XMPP stanzas.

A long-term storage key may be used to encrypt information stored in the key table or to re-encrypt encrypted messages prior to storing the message for future review. The long-term storage key may be a public/private key pair or a symmetric key.

5. Key table

The conceptual database for long-lived cryptographic keys described in [Key-Table] may be suitable for use in storing the SMKs described above for use in supporting end-to-end XMPP encryption. The columns that the table consists of are listed as follows:

TODO: figure out whether to read time values from JWKs. If so, augment section 8.2.

AdminKeyName: The AdminKeyName field contains a human-readable string meant to identify the key for the user. Implementations can use this field to uniquely identify rows in the key table. The same string can be used on the local system and peer systems, but this is not required.

LocalKeyName: The LocalKeyName field contains a string identifying the key. It can be used to retrieve the key in the local database when received in a message. For SMKs, this is the value of the 'id' attribute value of the <e2e/> element (see Section 6.3).

PeerKeyName: PeerKeyName is not used as the name is the same at each end point.

Peers: This field lists the full JID of each peer systems that has this key in their database. The peer name is read from the 'from' attribute of the wrapping stanza (see Section 6.3).

Interfaces: This field is not used and must be set to "all".

Protocol: The Protocol field identifies XMPP the protocol where this key may be used to provide cryptographic protection. (TODO: registry entry for the protocol?)

ProtocolSpecificInfo: This field is not used and must be empty.

KDF: The KDF field is not used and must be set to "none". (TODO: define a use for this field?)

AlgID: The AlgID field indicates which cryptographic algorithm to be used with the security protocol for the specified peer or peers. Such an algorithm can be an encryption algorithm and mode (e.g., AES-128-CBC), an authentication algorithm (e.g., HMAC-SHA1-96 or AES-128-CMAC), or any other symmetric cryptographic algorithm needed by a security protocol. (TODO: identify source for algorithm strings)

Key: The Key field contains a long-lived symmetric cryptographic key in the format of a lower-case hexadecimal string. The size of the Key depends on the KDF and the AlgID. For instance, a KDF=none and AlgID=AES128 requires a 128-bit key, which is represented by 32 hexadecimal digits.

Direction: The Direction field indicates whether this key may be used for inbound traffic, outbound traffic, both, or whether the key has been disabled and may not currently be used at all. The supported values are "in", "out", "both", and "disabled", respectively.

SendLifetimeStart: The SendLifetimeStart field specifies the earliest date and time in Coordinated Universal Time (UTC) at which this key should be considered for use when sending traffic. The format is YYYYMMDDHHSSZ, where four digits specify the year, two digits specify the month, two digits specify the day, two digits specify the hour, two digits specify the minute, and two digits specify the second. The "Z" is included as a clear indication that the time is in UTC.

SendLifeTimeEnd: The SendLifeTimeEnd field specifies the latest date and time at which this key should be considered for use when sending traffic. The format is the same as the SendLifetimeStart field.

AcceptLifeTimeStart: The AcceptLifeTimeStart field specifies the earliest date and time in Coordinated Universal Time (UTC) at which this key should be considered for use when processing received traffic. The format is YYYYMMDDHHSSZ, where four digits specify the year, two digits specify the month, two digits specify the day, two digits specify the hour, two digits specify the minute, and two digits specify the second. The "Z" is included as a clear indication that the time is in UTC.

AcceptLifeTimeEnd: The AcceptLifeTimeEnd field specifies the latest date and time at which this key should be considered for use when processing the received traffic. The format of this field is identical to the format of AcceptLifeTimeStart.

6. Encryption

6.1. Determining Support

If an agent supports receiving end-to-end object encryption, it **MUST** advertise that fact in its responses to [XEP-0030] information ("disco#info") requests by returning a feature of "urn:ietf:params:xml:ns:xmpp-e2e:6:encryption".

```
<iq xmlns='jabber:client'
  id='disco1'
  to='romeo@montegue.lit/garden'
  type='result'>
  <query xmlns='http://jabber.org/protocol/disco#info'>
    ...
    <feature xmlns='urn:ietf:params:xml:ns:xmpp-e2e:6:encryption' />
    ...
  </query>
</iq>
```

To facilitate discovery, an agent **SHOULD** also include [XEP-0115] information in any directed or broadcast presence updates.

6.2. Encrypting XMPP Stanzas

The process that a sending agent follows for securing stanzas is the same regardless of the form of stanza (i.e., <iq/>, <message/>, or <presence/>).

6.2.1. Prerequisites

First, the sending agent prepares and retains the following:

- o The JID of the sender (i.e. its own JID). This **SHOULD** be the bare JID (localpart@domainpart).
- o The JID of the recipient. This **SHOULD** be the bare JID (localpart@domainpart).
- o A Session Master Key (SMK). The SMK **MUST** have a length at least equal to that required by the key wrapping algorithm in use and **MUST** be generated randomly. See [RFC4086] for considerations on generating random values.

- o A SMK identifier (SID). The SID MUST be unique for a given (sender, recipient, SMK) tuple, and MUST NOT be derived from SMK itself.

6.2.2. Process

For a given plaintext stanza (S), the sending agent performs the following:

1. Ensures the plaintext stanza is fully qualified, including the proper namespace declarations (e.g., contains the attribute 'xmlns' set to the value "jabber:client" for 'jabber:client' stanzas defined in [RFC6120]).
2. Notes the current UTC date and time (N) when this stanza is constructed, formatted as described under Section 10.
3. Constructs a forwarding envelope (M) using a <forwarded/> element qualified by the "urn:xmpp:forward:0" namespace (as defined in [XEP-0297]) as follows:
 - * The child element <delay/> qualified by the "urn:xmpp:delay" namespace (as defined in [XEP-0203]) with the attribute 'stamp' set to the UTC date and time value N
 - * The plaintext stanza S
4. Converts the forwarding envelope (M) to a UTF-8 encoded string (M'), optionally removing line breaks and other insignificant whitespace between elements and attributes, i.e. M' = UTF8-encode(M). We call M' a "stanza-string" because for purposes of encryption and decryption it is treated not as XML but as an opaque string (this avoids the need for complex canonicalization of the XML input).
5. Generates a Content Master Key (CMK). The CMK MUST have a length at least equal to that required by the content encryption algorithm in use and MUST be generated randomly. See [RFC4086] for considerations on generating random values.

6. Generates any additional unprotected block cipher factors (IV); e.g., initialization vector/nonce. A sending agent MUST ensure that no two sets of factors are used with the same CMK, and SHOULD NOT reuse such factors for other stanzas.
7. Performs the message encryption steps from [JOSE-JWE] to generate the JWE Header (H), JWE Encrypted Key (E), JWE Ciphertext (C), and JWE Integrity Value (I); using the following inputs:
 - * The 'alg' property is set to an appropriate key wrapping algorithm (e.g., "A256KW" or "A128KW"); recipients use the key request process in Section 8 to obtain the SMK.
 - * The 'enc' property is set to the intended content encryption algorithm.
 - * SMK as the key for CMK Encryption.
 - * CMK as the JWE Content Master Key.
 - * IV as the JWE Initialization Vector.
 - * M' as the plaintext content to encrypt.
8. Constructs an <e2e/> element qualified by the "urn:ietf:params:xml:ns:xmpp-e2e:6" namespace as follows:
 - * The attribute 'type' set to the value "enc".
 - * The attribute 'id' set to the identifier value SID.
 - * The child element <encheader/> qualified by the "urn:ietf:params:xml:ns:xmpp-e2e:6" namespace and with XML character data as H, encoded base64url as per [RFC4648].

- * The child element `<cmk/>` qualified by the `"urn:ietf:params:xml:ns:xmpp-e2e:6"` namespace and with XML character as E, encoded base64url as per [RFC4648].
 - * The child element `<iv/>` qualified by the `"urn:ietf:params:xml:ns:xmpp-e2e:6"` namespace and with XML character as IV, encoded base64url as per [RFC4648].
 - * The child element `<data/>` qualified by the `"urn:ietf:params:xml:ns:xmpp-e2e:6"` namespace and with XML character data as C, encoded base64url as per [RFC4648].
 - * The child element `<mac/>` qualified by the `"urn:ietf:params:xml:ns:xmpp-e2e:6"` namespace and with XML character data as I, encoded base64url as per [RFC4648].
9. Sends the `<e2e/>` element as the payload of a stanza that SHOULD match the stanza from step 1 in kind (e.g., `<message/>`), type (e.g., "chat"), and addressing (e.g., `to="romeo@montague.net"` `from="juliet@capulet.net/balcony"`). If the original stanza (S) has a value for the 'id' attribute, this stanza MUST NOT use the same value for its 'id' attribute.

6.3. Decrypting XMPP Stanzas

6.3.1. Protocol Not Understood

If the receiving agent does not understand the protocol, it MUST do one and only one of the following: (1) ignore the `<e2e/>` extension, (2) ignore the entire stanza, or (3) return a `<service-unavailable/>` error to the sender, as described in [RFC6120].

NOTE: If the inbound stanza is an `<iq/>`, the receiving agent MUST return an error to the sending agent, to comply with the exchanging of IQ stanzas in [RFC6121].

6.3.2. Process

Upon receipt of an encrypted stanza, the receiving agent performs the following:

1. Determines if a valid SMK is available, associated with the SID specified by the 'id' attribute value of the `<e2e/>` element and

the sending agent JID specified by the 'from' attribute of the wrapping stanza. If the receiving agent does not already have the SMK, it requests it according to Section 8.

2. Performs the message decryption steps from [JOSE-JWE] to generate the plaintext forwarding envelope string M', using the following inputs:
 - * The JWE Header (H) from the <enheader/> element's character data content.
 - * The JWE Encrypted Key (E) from the <cmk/> element's character data content.
 - * The JWE Initialization Vector/Nonce (I) from the <iv/> element's character data content.
 - * The JWE Ciphertext (C) from the <data/> element's character data content.
 - * The JWE Integrity Value (I) from the <mac/> element's character data content.
3. Converts the forwarding envelope UTF-8 encoded string M' into XML element (M).
4. Obtains the UTC date and time (N) from the <delay/> child element, and verifies it is within the accepted range, as specified in Section 10.
5. Obtains the plaintext stanza (S), which is a child element node of M; the stanza MUST be fully qualified with proper namespace declarations for XMPP stanzas, to help distinguish it from other content within M.
- .

6.3.3. Insufficient Information

At step 1, if the receiving agent is unable to obtain the CMK, or the receiving agent could not otherwise determine the additional information, it MAY return a <bad-request/> error to the sending agent (as described in [RFC6120]), optionally supplemented by an application-specific error condition element of <insufficient-information/>:

```
<message xmlns='jabber:client'
  from='juliet@capulet.lit/balcony'
  id='fJZd9WFIIwNjFctT'
  to='romeo@montegue.lit/garden'
  type='chat'>
  <e2e xmlns='urn:ietf:params:xml:ns:xmpp-e2e:6'
    id='835c92a8-94cd-4e96-b3f3-b2e75a438f92'>
    <encheader>[XML character data]</encheader>
    <cmk>[XML character data]</cmk>
    <iv>[XML character data]</iv>
    <data>[XML character data]</data>
    <mac>[XML character data]</mac>
  </e2e>
  <error type='modify'>
    <bad-request
      xmlns='urn:ietf:params:xml:ns:xmpp-stanzas' />
    <insufficient-information
      xmlns='urn:ietf:params:xml:ns:xmpp-e2e:6' />
  </error>
</message>
```

In addition to returning an error, the receiving agent SHOULD NOT present the stanza to the intended recipient (human or application) and SHOULD provide some explicit alternate processing of the stanza (which MAY be to display a message informing the recipient that it has received a stanza that cannot be decrypted).

6.3.4. Failed Decryption

At step 2, if the receiving agent is unable to successfully decrypt the stanza, the receiving agent SHOULD return a <bad-request/> error to the sending agent (as described in [RFC6120]), optionally supplemented by an application-specific error condition element of <decryption-failed/> (previously defined in [RFC3923]):

```
<message xmlns='jabber:client'
  from='juliet@capulet.lit/balcony'
  id='fJZd9WFIIwNjFctT'
  to='romeo@montegue.lit/garden'
  type='chat'>
  <e2e xmlns='urn:ietf:params:xml:ns:xmpp-e2e:6'
    id='835c92a8-94cd-4e96-b3f3-b2e75a438f92'>
    <encheader>[XML character data]</encheader>
    <cmk>[XML character data]</cmk>
    <iv>[XML character data]</iv>
    <data>[XML character data]</data>
    <mac>[XML character data]</mac>
  </e2e>
  <error type='modify'>
    <bad-request xmlns='urn:ietf:params:xml:ns:xmpp-stanzas' />
    <decryption-failed xmlns='urn:ietf:params:xml:ns:xmpp-e2e:6' />
  </error>
</message>
```

In addition to returning an error, the receiving agent SHOULD NOT present the stanza to the intended recipient (human or application) and SHOULD provide some explicit alternate processing of the stanza (which MAY be to display a message informing the recipient that it has received a stanza that cannot be decrypted).

6.3.5. Timestamp Not Acceptable

At step 4, if the stanza is successfully decrypted but the timestamp fails the checks outlined in Section 10, the receiving agent MAY return a <not-acceptable/> error to the sender (as described in [RFC6120]), optionally supplemented by an application-specific error condition element of <bad-timestamp/> (previously defined in [RFC3923]):

```
<message xmlns='jabber:client'
  from='juliet@capulet.lit/balcony'
  id='fJZd9WFIIwNjFctT'
  to='romeo@montegue.lit/garden'
  type='chat'>
  <e2e xmlns='urn:ietf:params:xml:ns:xmpp-e2e:6'
    id='835c92a8-94cd-4e96-b3f3-b2e75a438f92'>
    <encheader>[XML character data]</encheader>
    <cmk>[XML character data]</cmk>
    <iv>[XML character data]</iv>
    <data>[XML character data]</data>
    <mac>[XML character data]</mac>
  </e2e>
  <error type='modify'>
    <bad-request xmlns='urn:ietf:params:xml:ns:xmpp-stanzas' />
    <bad-timestamp xmlns='urn:ietf:params:xml:ns:xmpp-e2e:6' />
  </error>
</message>
```

6.3.6. Successful Decryption

If the receiving agent successfully decrypted the payload, it MUST NOT return a stanza error.

If the payload is an <iq/> of type "get" or "set", and the response to this <iq/> is of type "error", the receiving agent MUST send the encrypted response wrapped in an <iq/> of type "result", to prevent exposing information about the payload.

6.4. Example - Securing a Message

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

The sending agent begins with the plaintext version of the <message/> stanza 'S':

```
<message xmlns='jabber:client'
  from='juliet@capulet.lit/balcony'
  to='romeo@montegue.lit'
  type='chat'>
  <thread>35740be5-b5a4-4c4e-962a-a03b14ed92f4</thread>
  <body>
    But to be frank, and give it thee again.
    And yet I wish but for the thing I have.
    My bounty is as boundless as the sea,
    My love as deep; the more I give to thee,
    The more I have, for both are infinite.
  </body>
</message>
```

and the following prerequisites:

- o Sender JID as "juliet@capulet.lit/balcony"
- o Recipient JID as "romeo@montegue.lit"
- o Session Master Key (SMK) as (base64 encoded)
"xWtdjhYsH4Va_9SfYSefsJfZu03m5RrbXo_UavxxeU8"
- o SMK identifier (SID) as "835c92a8-94cd-4e96-b3f3-b2e75a438f92"

The sending agent performs steps 1, 2, and 3 from Section 6.2.2 to generate the envelope:

```
<forwarded xmlns='urn:xmpp:forward:0'>
  <delay xmlns='urn:xmpp:delay'
    stamp='1492-05-12T20:07:37.012Z' />
  <message xmlns='jabber:client'
    from='juliet@capulet.lit/balcony'
    to='romeo@montague.lit'
    type='chat'>
    <thread>35740be5-b5a4-4c4e-962a-a03b14ed92f4</thread>
    <body>
      But to be frank, and give it thee again.
      And yet I wish but for the thing I have.
      My bounty is as boundless as the sea,
      My love as deep; the more I give to thee,
      The more I have, for both are infinite.
    </body>
  </message>
</forwarded>
```

Then the sending agent performs steps 4 through 7 (with Content Master Key as "LViSXX0Jx-I3vlzY1-KcGeivmWKuq0QE_7lywQGU6OhlM2NoQolzhI77zI3ieIUh7Wb1S3kXmNily0_FZoIG7A", base64url encoded) to generate the [JOSE-JWE] outputs:

JWE Header

```
{
  "alg": "A256KW",
  "enc": "A256CBC+HS512",
  "kid": "835c92a8-94cd-4e96-b3f3-b2e75a438f92"
}
```

JWE Encrypted Key

```
2tsmGH-WQdBxxJES3d6LB2ovK6e1_9ClogizJ9c6OvLmC6IeilHZ2Mimq2AEIgI
ploz0VQv5LOH9ST93WvvhVzMHSfx0Cwl0
```

JWE Initialization Vector

```
ncOH4MsHT9HlJxnirx4qwg
```

JWE Ciphertext

FkFc4xGTVkjn7oJtS0SUY8IWfqsQKEIAlvLaBKieqVX1PA1q1ZjPp4TZC2I2eh7
01Lef3iRuNZd1nlgP2aREyHYCPe3FAelUoVG90B1FrJMnDUKAka7eb6GImamWPf
9onV-m5-GcUpejO9f1oPi-rwHzp475UPdAeKq5Z4zds8yXhQP-XyJbCPTtM-UQC
2-_q-3EKBHC4jM3qWDxVJ0JbIif3fCVRowzJh4AOB84YrfvkgUjMITqQPg2H6QB
NqGUspLI6341M8R-mhGciDZX2Jh_nKoXLaf5GCnvL9PlI7OdFqocPBIIpJNrgX
_Z4PFjeq7ILx98GhVkryLYU9HVOFPCYci-lF9nfwlgeliLfkoj5QZyi4J2S0tYa
O_zPmQvCXaUREqPf5UDAlgvc50a4ByYnNbKWSbhZ5Z388s8ELzPSE9XypdgP-1c
SyRke7V8iGe4eHNsm01TgWILYOFK4mYAM52OTitJxmQtmRp6izY5ZFdH9f_WdoB
1RXmGEZydvL-estcJx5ghsV3gktdIl0HA4R_M_N5TFIwv7hiisyRLi2aQtyFbE
7pZ6Oz-cYsLc4qFfXbb13U9a2-Byul8hm_E2b3m4GMhmsCiROm-uht9Ek4h9Bix
FhDKPr-htOXc93-uQNZlAQfkITAKlJfQ

JWE Integrity Value

Aj8lkdPMDE4U82UAhDJBaRrl3USmuzS2hfFOe_OBEv8

Then the sending agent performs steps 8 and 9, and sends the following:

```

<message xmlns='jabber:client'
  from='juliet@capulet.lit/balcony'
  id='fJZd9WFIIwNjFctT'
  to='romeo@montegue.lit'
  type='chat'>
  <e2e xmlns='urn:ietf:params:xml:ns:xmpp-e2e:6'
    type='enc'
    id='835c92a8-94cd-4e96-b3f3-b2e75a438f92'>
    <encheader>
      eyJhbGciOiJBbmJlLCJlbmMiOiJBbmJlU2Q0JDK0hTNTEyIiwia2lkI
      joiodMlYzkyYTgtOTRjZC00ZTk2LWlWizZjMtYjJlNzVhNDM4ZjkyIn0
    </encheader>
    <cmk>
      2tsmGH-WQdBxxJEs3d6LB2ovK6e1_9ClogizJ9c6OvLmC6IeilHZ2Mimq
      2AElgIploz0VQv5LOH9ST93WvvhVzMHSfx0Cw10
    </cmk>
    <iv>
      ncOH4MsHT9HlJxnirx4qwg
    </iv>
    <data>
      FkFc4xGTVk jn7ojtS0SUY8IWfqsQKEIAlvLaBKieqVX1PA1q1zjPp4TZC
      2I2eh701Lef3iRuNZd1nlGP2aREyHYCPe3FAelUoVG90B1FrJMnDUKAKa
      7eb6GImamWPf9onV-m5-GcUpejO9floPi-rwHzp475UPdAeKq5Z4zds8y
      XhQP-XyJbCPTtM-UQC2-_q-3EKBHC4jM3qWDxVJ0JbIif3fCVRowzJh4A
      OB84YrfvkgUjMitqQPg2H6QBNqGUSpLI634lM8R-mhGciDZX2Jh_nKoXL
      Af5GcNvL9PlI7OdFqocPBIIppjNrgX_Z4PFjeq7ILx98GhVkrYLYU9HVO
      FPCYci-lF9nfwlgeliLfkoj5QZyi4J2SotYaO_zPmQvCXaUREqPf5UDAl
      gvc50a4ByYnNbKWSbhZ5Z388s8ELzPSE9XypdgP-1cSyRke7V8iGe4eHN
      sm01TgWILYOFK4mYAM52OTitJxmQtmRp6izY5ZFdh9f_WdoB1RXmGEZyd
      vL-estc jx5ghsV3gktedIl0HA4R_M_N5TFIwv7hiisyRLi2aQtyFbE7pZ
      6Oz-cYsLc4qFfXbb13U9a2-Byul8hm_E2b3m4GMhmsCiROm-uht9Ek4h9
      BixFhDKPr-htOXc93-uQNZlAQfkITAKlJfQ
    </data>
    <mac>
      Aj8lKdPMDE4U82UAhDJBaRrl3USmuzS2hffOe_OBEv8
    </mac>
  </e2e>
</message>

```

7. Signatures

7.1. Determining Support

If an agent supports receiving end-to-end object signatures, it **MUST** advertise that fact in its responses to [XEP-0030] information ("disco#info") requests by returning a feature of "urn:ietf:params:xml:ns:xmpp-e2e:6:signatures".

```
<iq xmlns='jabber:client'
  id='disco1'
  to='romeo@montegue.lit/garden'
  type='result'>
  <query xmlns='http://jabber.org/protocol/disco#info'>
    ...
    <feature xmlns='urn:ietf:params:xml:ns:xmpp-e2e:6:signatures' />
    ...
  </query>
</iq>
```

To facilitate discovery, an agent SHOULD also include [XEP-0115] information in any directed or broadcast presence updates.

7.2. Signing XMPP Stanzas

The basic process that a sending agent follows for authenticating stanzas is the same regardless of the kind of stanza (i.e., <iq/>, <message/>, or <presence/>).

7.2.1. Process

For a given plaintext stanza (S), the sending agent performs the following:

1. Ensures the plaintext stanza is fully qualified, including the proper namespace declarations (e.g., contains the attribute 'xmlns' set to the value "jabber:client" for 'jabber:client' stanzas defined in [RFC6120]).
2. Notes the current UTC date and time (N) when this stanza is constructed, formatted as described under Section 10.
3. Constructs a forwarding envelope (M) using a <forwarded/> element qualified by the "urn:xmpp:forward:0" namespace (as defined in [XEP-0297]) as follows:

- * The child element <delay/> qualified by the "urn:xmpp:delay" namespace (as defined in [XEP-0203]) with the attribute 'stamp' set to the UTC date and time value N

- * The plaintext stanza S

4. Converts the forwarding envelope (M) to a UTF-8 encoded string (M'), optionallly removing line breaks and other insignificant whitespace between elements and attributes, i.e. M' = UTF8-encode(M). We call M' a "stanza-string" because for purposes of encryption and decryption it is treated not as XML but as an opaque string (this avoids the need for complex canonicalization of the XML input).
5. Chooses a private asymmetric key (PK) for which the sending agent has published the corresponding public key to the intended recipients.
6. Performs the message signatures steps from [JOSE-JWS] to generate the JWS Header (H) and JWS Signature (I); using the following inputs:
 - * The 'alg' property is set to an appropriate signature algorithm for PK (e.g., "R256").
 - * M' as the JWS Payload.
7. Constructs an <e2e/> element qualified by the "urn:ietf:params:xml:ns:xmpp-e2e:6" namespace as follows:
 - * The attribute 'type' set to the value "sig"
 - * The child element <sigheader/> qualified by the "urn:ietf:params:xml:ns:xmpp-e2e:6" namespace and with XML character data as H, encoded base64url as per [RFC4648].
 - * The child element <data/> qualified by the "urn:ietf:params:xml:ns:xmpp-e2e:6" namespace and with XML character data as M', encoded base64url as per [RFC4648].
 - * The child element <sig/> qualified by the "urn:ietf:params:xml:ns:xmpp-e2e:6" namespace and with XML character data as I, encoded base64url as per [RFC4648].

8. Sends the <e2e/> element as the payload of a stanza that SHOULD match the stanza from step 1 in kind (e.g., <message/>), type (e.g., "chat"), and addressing (e.g., to="romeo@montegue.lit" from="juliet@capulet.lit/balcony"). If the original stanza (S) has a value for the 'id' attribute, this stanza SHOULD NOT use the same value for its "id" attribute.

7.3. Verifying Signed XMPP Stanzas

7.3.1. Protocol Not Understood

If the receiving agent does not understand the protocol, it MUST do one and only one of the following: (1) ignore the <e2e/> extension, (2) ignore the entire stanza, or (3) return a <service-unavailable/> error to the sender, as described in [RFC6120].

NOTE: If the inbound stanza is an <iq/>, the receiving agent MUST return an error to the sending agent, to comply with the exchanging of IQ stanzas in [RFC6121].

7.3.2. Process

Upon receipt of a signed stanza, the receiving agent performs the following:

1. Ensures it has appropriate materials to verify the signature, which generally means ensuring that it possesses one or more public keys for the sending agent (if one is not provided as part of the JWS Header).
2. Performs the message validation steps from [JOSE-JWS], with the following inputs:
 - * The JWS Header H from the <sigheader/> element's character data content.
 - * The JWS payload M' from the <data/> element's character data content.
 - * The JWS Signature from the <sig/> element's character data content.

3. Converts the forwarding envelope UTF-encoded string M' into XML element M.
4. Obtains the UTC date and time N from the <delay/> child element, and verifies it is within the accepted range, as specified in Section 10.
5. Obtains the plaintext stanza S, which is a child element node of M; the stanza MUST be fully qualified with the proper namespace declarations from XMPP stanzas, to help distinguish it from other content within M.

7.3.3. Insufficient Information

At step 1, if the receiving agent does not have the key used to sign the stanza, or the receiving agent could not otherwise determine it, it MAY return a <bad-request/> error to the sending agent (as described in [RFC6120]), optionally supplemented by an application-specific error condition element of <insufficient-information/>:

```
<message xmlns='jabber:client'
  from='juliet@capulet.lit/balcony'
  id='fJZd9WFIIwNjFctT'
  to='romeo@montague.lit/garden'
  type='chat'>
  <e2e xmlns='urn:ietf:params:xml:ns:xmpp-e2e:6'
    type='sig'>
    <sigheader>[XML character data]</sigheader>
    <data>[XML character data]</data>
    <sig>[XML character data]</sig>
  </e2e>
  <error type='modify'>
    <bad-request
      xmlns='urn:ietf:params:xml:ns:xmpp-stanzas' />
    <insufficient-information
      xmlns='urn:ietf:params:xml:ns:xmpp-e2e:6' />
  </error>
</message>
```

In addition to returning an error, the receiving agent SHOULD NOT present the stanza to the intended recipient (human or application) and SHOULD provide some explicit alternate processing of the stanza (which MAY be to display a message informing the recipient that it has received a stanza that cannot be verified).

7.3.4. Failed Verification

At step 2, if the receiving agent is unable to successfully verify the stanza, the receiving agent SHOULD return a <bad-request/> error to the sending agent (as described in [RFC6120]), optionally supplemented by an application-specific error condition element of <verification-failed/>:

```
<message xmlns='jabber:client'
  from='juliet@capulet.lit/balcony'
  id='fJZd9WFIIwNjFctT'
  to='romeo@montegue.lit/garden'
  type='chat'>
  <e2e xmlns='urn:ietf:params:xml:ns:xmpp-e2e:6'
    type='sig'>
    <sigheader>[XML character data]</sigheader>
    <data>[XML character data]</data>
    <sig>[XML character data]</sig>
  </e2e>
  <error type='modify'>
    <bad-request
      xmlns='urn:ietf:params:xml:ns:xmpp-stanzas' />
    <verification-failed
      xmlns='urn:ietf:params:xml:ns:xmpp-e2e:6' />
  </error>
</message>
```

In addition to returning an error, the receiving agent SHOULD NOT present the stanza to the intended recipient (human or application) and SHOULD provide some explicit alternate processing of the stanza (which MAY be to display a message informing the recipient that it has received a stanza that cannot be verified).

7.3.5. Timestamp Not Acceptable

At step 4, if the stanza is successfully verified but the timestamp fails the checks outlined in Section 10, the receiving agent MAY return a <not-acceptable/> error to the sender (as described in [RFC6120]), optionally supplemented by an application-specific error condition element of <bad-timestamp/> (previously defined in [RFC3923]):

```
<message xmlns='jabber:client'
  from='juliet@capulet.lit/balcony'
  id='fJZd9WFIIwNjFctT'
  to='romeo@montegue.lit/garden'
  type='chat'>
  <e2e xmlns='urn:ietf:params:xml:ns:xmpp-e2e:6'
    type='sig'>
    <sigheader>[XML character data]</sigheader>
    <data>[XML character data]</data>
    <sig>[XML character data]</sig>
  </e2e>
  <error type='modify'>
    <not-acceptable
      xmlns='urn:ietf:params:xml:ns:xmpp-stanzas' />
    <bad-timestamp
      xmlns='urn:ietf:params:xml:ns:xmpp-e2e:6' />
  </error>
</message>
```

7.3.6. Successful Verification

If the receiving agent successfully verified the payload, it SHOULD NOT return a stanza error. However, if the signed stanza is an <iq/> of type "get" or "set", the response MAY be sent unsigned if the receiving agent does not have an appropriate public-private key-pair.

Otherwise, the receiving agent SHOULD send the <iq/> response signed as per Section 7.2.1, with the 'type' attribute set to the value "result", even if the response to the signed <iq/> stanza is of type "error". The error applies to the signed stanza, not the wrapping stanza.

7.4. Example - Signing a Message

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

The sending agent begins with the plaintext version of <message/> stanza 'S':

```
<message xmlns='jabber:client'
  from='juliet@capulet.lit/balcony'
  to='romeo@montegue.lit'
  type='chat'>
  <thread>35740be5-b5a4-4c4e-962a-a03b14ed92f4</thread>
  <body>
    But to be frank, and give it thee again.
    And yet I wish but for the thing I have.
    My bounty is as boundless as the sea,
    My love as deep; the more I give to thee,
    The more I have, for both are infinite.
  </body>
</message>
```

Then the sending agent performs steps 1, 2, and 3 from Section 7.2.1 generate the envelope M:

```
<forwarded xmlns='urn:xmpp:forward:0'>
  <delay xmlns='urn:xmpp:delay'
    stamp='1492-05-12T20:07:37.012Z' />
  <message xmlns='jabber:client'
    from='juliet@capulet.lit/balcony'
    to='romeo@montegue.lit'
    type='chat'>
    <thread>35740be5-b5a4-4c4e-962a-a03b14ed92f4</thread>
    <body>
      But to be frank, and give it thee again.
      And yet I wish but for the thing I have.
      My bounty is as boundless as the sea,
      My love as deep; the more I give to thee,
      The more I have, for both are infinite.
    </body>
  </message>
</forwarded>
```

Then the sending agent performs steps 4, 5, and 6 to generate the [JOSE-JWS] outputs:

JWS Header (before base64url encoding)

```
{
  "alg": "RS512",
  "kid": "juliet@capulet.lit"
}
```

JWS Payload

PGZvcndhcmRlZCB4bWxucz0idXJuOnhtcHA6Zm9yd2FyZDowIj48ZGVsYXkgeG1
sbnM9InVybjp4bXBwOmRlbGF5IiBzdGFtcD0iMTQ5Mi0wNS0xMlQyMDowNzozNy
4wMTJaIi8-PG1lc3NhZ2UgeG1sbnM9ImphYmJlcjpbGllbnQiIGZyb209Imp1b
GlldeBjYXB1bGV0LmxpdC9iYWxjb255IiB0bz0icm9tZW9AbW9udGVndWUubG10
IiB0eXB1PSJjaGF0Ij48dGhyZWFKPjM1NzQwYmU1LWI1YTQtNGM0ZS05NjJhLWE
wM2IxNGVkOTJmNDwvdGhyZWFKPjxib2R5PkJldCB0byBiZSBmcmFuaywgYW5kIG
dpdmUgaXQgdGhlZSBhZ2Fpbi4gQW5kIHl1dCBJIHdpc2ggYnV0IGZvciB0aGUgd
GhpbmGSSBoYXZlLiBNeSBib3VudHkgXMGYXMGYm91bmRsZXNzIGFzIHROZSBz
ZWEsIE15IGxvdmUgYXMGZGVlcDsgdGhlIGlvcuUgSSBnaXZlIHRvIHROZWUsIFR
oZSBtb3JlIEkgaGF2ZSwgZm9yIGJvdGggYXJlIGluZmluaXRlLjwvYm9keT48L2
1lc3NhZ2U-PC9mb3J3YXJkZWQ-

JWS Signature

YPfGouD50j0C_C-RneawG0jxXWDXgBkN3FJz6eaBFIPCh3hopiwtwKir7Yamvgt
OrqhXx2pcu-70caGi6mKKLWvpdwdJ3nEnhdjPOd3CmLdaK_PBAMtIt8d3155hdl
qNxSMsJN7PxmNLNwJhbksAsI-2TcCQsuxdIPXh6hcqBm44BpVio6AoRPqWf06XZ
MMBMOMnEFcV6Ht20wCK1BEGgOmN3KYPbwKeTctG8HKPAh25_K66aEXT66lI19uW
jlfGFJ79QQHUhc5y9pSKmpK7HKruPMRyrvpzBSfUhcb62nLXhM-LzY5taaDECzi
fCi-IxySBtJJtPCqYAYW_IbrRFg

Then the sending agent performs steps 7 and 8 and sends the following:

```

<message xmlns='jabber:client'
  from='juliet@capulet.lit/balcony'
  id='6aAWpciGV98qaegk'
  to='romeo@montegue.lit'
  type='cat'>
  <e2e xmlns='urn:ietf:params:xml:ns:xmpp-e2e:6'
    type='sig'>
    <sigheader>
      eyJhbGciOiJSUzUxMiIsImtpZCI6ImplbGlldEBjYXB1bGV0LmxpdCJ9
    </sigheader>
    <data>
      PGZvcndhcmRlZCB4bWxucz0idXJuOnhtcHA6Zm9yd2FyZDowIj48ZGVsY
      XkgeGlsbnM9InVybjp4bXBwOmRlbGF5IiBzdGFtcD0iMTQ5Mi0wNS0xMl
      QyMDowNzozNy4wMTJaIi8-PG1lc3NhZ2UgeGlsbnM9ImphYmJlcjpbG1
      lbnQiIGZyb209ImplbGlldEBjYXB1bGV0LmxpdC9iYWxjb255IiB0bz0i
      cm9tZW9AbW9udGVndWUubGl0IiB0eXB1PSJjaGF0Ij48dGhyZWFKPjM1N
      zQwYmU1LWI1YTQtNGM0ZS05NjJhLWEwM2IxNGVkotJmNDwvdGhyZWFKPj
      xib2R5PkJldCB0byBiZSBmcmFuaywgYW5kIGdpdmUgaXQgdGhlZSBhZ2F
      pbi4gQW5kIHllZCBJIHdpc2ggYnV0IGZvciB0aGUgdGhpbmCGSSBoYXZl
      LiBNeSBib3VudHkgaXMgYXMGYm91bmRsZXNzIGFzIHROZSBzZWESIE15I
      GxvdmUgYXMGZGVlcDsgdGhlIGlvcmluZSBBnaXZlIHVhZDZlZWU5IFRoZS
      Btb3JlIEkgaGF2ZSwgZm9yIGJvdGggYXJlIGluZmluaXRlLjwvYm9keT4
      8L21lc3NhZ2U-PC9mb3J3YXJkZWQ-
    </data>
    <sig>
      YPfgouD50j0C_C-RneawG0jxXWDXgBkN3FJz6eaBFIPCh3hopiwtwKir7
      YamvgtoRqhXx2pcu-70caGi6mKKLWvpdwdJ3nEnhdjPOd3CmLdaK_PBAM
      tIt8d3155hdlqNxSMsJN7PxmNLNwJhbksAsI-2TcCQsuxdIPXh6hcqBm4
      4BpVio6AoRPqwf06XZMMBMOMnEFcV6Ht20wCK1BEGGomN3KYPbwKeTctG
      8HKPAh25_K66aEXT66lI19uWj1fGFJ79QQHUhc5y9pSKmpK7HKruPMRyr
      vpzBSfUhc62nLXhM-LzY5taaDECzifCi-IxySBtJtPCqYAYW_IbrRFg
    </sig>
  </e2e>
</message>

```

8. Requesting Session Keys

Because of the dynamic nature of XMPP stanza routing, the protocol does not exchange session keys as part of the encrypted stanza. Instead, a separate protocol is used by receiving agents to request a particular session key from the sending agent.

8.1. Request Process

Before a SMK can be requested, the receiving agent **MUST** have at least one public key for which it also has the private key. The public key(s) are provided to the sending agent as part of this process.

To request a SMK, the receiving agent performs the following:

1. Constructs a [JOSE-JWK] JWK Set (KS), containing information about each public key the requesting agent wishes to use. Each key SHOULD include a value for the property 'kid' which uniquely identifies it within the context of all provided keys. Each key MUST include a value for the property 'kid' if any two keys use the same algorithm.
2. Constructs a <keyreq/> element qualified by the "urn:ietf:params:xml:ns:xmpp-e2e:6" namespace as follows:
 - * The attribute 'id' set to the SMK identifier value SID.
 - * The child element <pkey/> qualified by the "urn:ietf:params:xml:ns:xmpp-e2e:6" namespace and with XML character data as KS, encoded base64url as per [RFC4648].
3. Sends the <keyreq/> element as the payload of an <iq/> stanza with the attribute 'type' set to "get", the attribute 'to' set to the full JID of the original encrypted stanza's sender, and the attribute 'id' set to an opaque string value the receiving agent uses to track the <iq/> response.

8.2. Accept Process

If the sending agent approves the request, it performs the following steps:

1. Generate a JSON Web Key (JWK) representing the symmetric SMK (according to [JOSE-JWK]):
 - * The "kty" parameter MUST be "oct".
 - * The "kid" parameter MUST be the SID.

- * The "k" parameter MUST be the SMK, encoded as base64url.
 - * The "alg" parameter, if present, MUST be set to the algorithm in use for encrypting messages from Section 6.2.
 - * The "use" parameter, if present, MUST be set to "enc".
2. Chooses a key (PK) from the keys provided via KS, and notes its identifier value 'kid'.
 3. Protects the SMK using the process outlined in [JOSE-KEYPROTECT] to generate the JWE Header (H), JWE Encrypted Key (E), JWE Initialization Vector (IV), JWE Ciphertext (C), and JWE Integrity Value (I); using the following inputs:
 - * The 'alg' property is set to an algorithm appropriate for the chosen PK (e.g., "RSA-OAEP" for a "RSA" key).
 - * The 'enc' property is set to the intended content encryption algorithm.
 - * A randomly generated CMK. See [RFC4086] for considerations on generating random values.
 - * A randomly generated initialization vector. See [RFC4086] for considerations on generating random values.
 - * SMK, formatted as a JWK as above.
 4. Constructs a <keyreq/> element qualified by the "urn:ietf:params:xml:ns:xmpp-e2e:6" namespace as follows:
 - * The attribute 'id' set to the SMK Identifier (SID).

- * The child element <encheader/> qualified by the "urn:ietf:params:xml:ns:xmpp-e2e:6" namespace and with XML character data as H, encoded base64url as per [RFC4648].
 - * The child element <cmk/> qualified by the "urn:ietf:params:xml:ns:xmpp-e2e:6" namespace and with XML character data as E, encoded base64url as per [RFC4648].
 - * The child element <iv/> qualified by the "urn:ietf:params:xml:ns:xmpp-e2e:6" namespace and with XML character data as IV, encoded base64url as per [RFC4648].
 - * The child element <data/> qualified by the "urn:ietf:params:xml:ns:xmpp-e2e:6" namespace and with XML character data as C, encoded base64url as per [RFC4648].
 - * The child element <mac/> qualified by the "urn:ietf:params:xml:ns:xmpp-e2e:6" namespace and with XML character data as I, encoded base64url as per [RFC4648].
5. Sends the <keyreq/> element as the payload of an <iq/> stanza with the attribute 'type' set to "result", the attribute 'to' set to the full JID from the request <iq/>'s 'from' attribute, and the attribute 'id' set to the value of the request <iq/>'s 'id' attribute.

8.3. Error Conditions

If the sending agent does not approve the request, it sends an <iq/> stanza of type "error" and containing the reason for denying the request:

- o <forbidden/>: the key request is made by an entity that is not authorized to decrypt stanzas from the sending agent and/or for the indicated SID.
- o <item-not-found/>: the requested SID is no longer valid.

- o <not-acceptable/>: the key request did not contain any keys the sending agent understands.

8.4. Example of Successful Key Request

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

To begin a key request, the receiving agent performs step 1 from Section 8.1 to generate the [JOSE-JWK]:

```
{
  "keys": [{
    "kty": "RSA",
    "kid": "romeo@montegue.lit/garden",
    "n": "vtqejkMF01h8oKEaHfHEY00C2jM7eISbbSvNs0SNItYW06GbJpJf
N4ldXw2vpVRdysnwU3zk6o2_SD0YCHlWgeuI0QK1knMTDdNSXx52elc4BTw
hlA8iHuutTWmpBquesn1GNZmqB3jYsJOkVBYwCJtkB9APaBvk0itlRtizjCf
1HHnau7nGStyshgu8-srxi_d8rC5TTLsb_zTli6fP8fwdloemX0tC0U65by
5P-1ZHXaf_bd8fpjps6gwSgdkZKMJAI0bOWZWuMpp2ntqa0wLB7Ndx2Ijr
eog_s5ssAoSiXDVDoswSbp36ZP-1lnCk2j-vZ4qbhaFg5bZtgt-gwQ",
    "e": "AQAB"
  }]
}
```

Then the receiving agent performs step 2 to generate the <keyreq/>:

```
<keyreq xmlns='urn:ietf:params:xml:ns:xmpp-e2e:6'
  id='835c92a8-94cd-4e96-b3f3-b2e75a438f92'>
  <pkey>
    eyJrZXlziJpbeyJrdHkiOiJSU0EiLCJraWQiOiJyb21lb0Btb250ZWdlZS5
    saXQvZ2FyZGVuIiwib21lb0Btb250ZWdlZS5saXQvZ2FyZGVuIiwib21lb0Btb250ZWdlZS5
    d1SVNiYlN2TnMwU05JdFlXTzZHYmpwSmZONGxkWHcydnBWUmR5c253VTN6a
    zZvMl9TRDBZQ0gxV2dldUkwUUsxa25NVERkTlNYeDUyZTFjNEJUd2hsQTlp
    SHVldFRXbXBCCWVzbjFHTlptcUIzallzSk9rVkJZd0NKdGtCOUFQYUJ2azB
    pdGxSdG16akNmMUhIbmFlN25HU3R5c2hndTgtc3J4aV9kOHJDNRVUTFNCR3
    pUMWk2ZlA4ZndEbG9lbVhPdEMwVTY1Ynk1UC0xWkh4YWZfYkQ4ZnBqcHM2Z
    3dTZ2RrWktNSkFJMGJpV1pXdu1wcDJudHhMHdMQjdOZHhiMklqcmVvZ19z
    NXNzQW9TaVhEVmRvc3dTYnAzNlpQLTFsbkNrMmotdlo0cWJoYUZNWJadGd
    0LWd3USIsImUiOiJBUUFClndfQ
  </pkey>
</keyreq>
```

Then the receiving agent performs step 3 and sends the following:

```

<iq xmlns='jabber:client'
  from='romeo@montegue.lit/garden'
  id='xdJbWMA+'
  to='juliet@capulet.lit/balcony'
  type='get'>
  <keyreq xmlns='urn:ietf:params:xml:ns:xmpp-e2e:6'
    id='835c92a8-94cd-4e96-b3f3-b2e75a438f92'>
    <pkey>
      eyJrZXlzlIjpbeyJrdHkiOiJSU0EiLCJraWQiOiJyb21lb0Btb250ZWd1ZS5saXQvZ2FyZGVuIiwibiI6InZ0cWVqa01GMDFoOG9LRWFIZkhFWU8wQzJqTTdlSVNiYlN2TnMwU05JdFlXTZzZHYmpwSmZONGxkWHcydnBWUmR5c253VTN6azZvMl9TRDBZQ0gxV2dldUkwUUsxa25NVERkTlNYeDUyZTFjNEJUd2hsQThtpSHVldFRXbXBCCwVzbjFHTlptcUIzallzSk9rVkJZd0NKdGtCOUFQYUJ2azBpdGxSdGl6akNmMUhIbmF1N25HU3R5c2hndTgtc3J4aV9kOHJDNVRUTFNCX3pUMWk2ZlA4ZndEbG9lbVhPdEMwVTY1Ynk1UC0xWkh4YWZfYkQ4ZnBqcHM2Z3dTZ2RrWktNSkFJMGJpV1pXdlwlcDJudHFhMHdMQjdOZHhiMklqcmVvZ19zNXNzQW9TaVhEVmRvc3dTlYnAzNlpQLTFsbkNrMmotdlo0cWJoYUZNWJadGd0LWd3USIsImUiOiJBUUFFCinldfQ
    </pkey>
  </keyreq>
</iq>

```

If the sending agent accepts this key request, it performs step 1 from Section 8.2 to generate JWK representation of the SMK:

```

{
  "kty": "oct",
  "kid": "835c92a8-94cd-4e96-b3f3-b2e75a438f92",
  "k": "xWtdjhYsh4Va_9SfYSefsJfZu03m5RrbXo_UavxxeU8"
}

```

Then the sending agent performs steps 2 and 3 to generate the protected SMK:

JWE Header (before base64url encoding)

```

{
  "alg": "RSA-OAEP",
  "kid": "romeo@montegue.lit/garden",
  "enc": "A256CBC+HS512",
  "cty": "application/jwk+json"
}

```

JWE Encrypted Key

hKUOpAif76c-hmRwEphVB9wXjloLpwu75x98MSWyCBtfUgmopk93ttUXoZ4AAIk
rZJOtrPUqPZwYHJay3ggfgjVljJ_KGhgqI5cScIzaAQs0Pxep6FnrsnUrw09Sjv
2VRXOay4guMQnbQo0ibpifBxeuL9MJ_vdeb_BdSE8YZ4iTfMb7GT35gZC9NgweX
3fiTEo2LjY8hEV3DHud5LlNZzYp9kLmAUZNIwGu7LtYyI4F7NnOv9oLx1HtmfE3
_skkYtQoKMvMewLkIO88h325qCpWFdrLwPp63betCmewDJPaBdrp91rLchkXVo-
d2ueKkb59TxWjMx7esBdaxCAcDQ

JWE Initialization Vector

Ggiego8UiSsj7GgY94qOng

JWE Ciphertext

4vIGDz9Hm6X4lSo9JoA6ZzS0KitztLGAiMUs3RTviFO09choPhxJNlOj8KX8QIL
u4zZ-ytCnG-yzNx5SsT8KEQJhIf6_9yWplxpXl73k6ZJV-sXGd4Mj9u7N0IqWQL
K5DMytv7XopsZsR9QFCDNGew

JWE Integrity Value

3GuaasWV0XGTBbRtNP6OQ14_cHL-ZJClnaDtU6EIecw

Then the sending agent performs step 4 to generate the <keyreq/>
response:

```
<keyreq xmlns='urn:ietf:params:xml:ns:xmpp-e2e:6'
  id='835c92a8-94cd-4e96-b3f3-b2e75a438f92'>
  <encheader>
    eyJhbGciOiJSU0EtT0FFUCIsImtpZCI6InJvbWVvQG1vbnRlZ3VlLmexpdC9
    nYXJkZW4iLCJlbmMiOiJBMjU2Q0JDK0hTNTYiIiwia3R5IjoieXBwbGljYX
    Rpb24vandrK2pzb24ifQ
  </encheader>
  <cmk>
    hKUOpAif76c-hmRwEphVB9wXjloLpwu75x98MSWyCBtfUgmopk93ttUXoZ4
    AAIkrZJOtrPUqPZwYHjay3ggfgjVljJ_KGhgqI5cScIzaAQs0Pxep6Fnrsn
    Urw09Sjv2VRXOay4guMQnbQo0ibpifBxeuL9MJ_vdeb_BdSE8YZ4iTfMb7G
    T35gZC9NgweX3fiTEo2LjY8hEV3DHud5LlNZzYp9kLmAUZNIwGu7LtYyI4F
    7NnOv9oLx1HtmfE3_skkYtQoKMvMewLkIO88h325qCpWFdrLwPp63betCme
    wDJPaBdrp91rLchkXVo-d2ueKkb59TxWjMx7esBdaxCAcDQ
  </cmk>
  <iv>
    Ggiego8UiSsj7GgY94qOng
  </iv>
  <data>
    4vIGDz9Hm6X4lSo9JoA6ZzS0KitztLGaiMUs3RTviFO09choPhxJNlOj8KX
    8QILu4zZ-ytCnG-yzNx5SsT8KEQJhIf6_9yWplxpXl73k6ZJV-sXGd4Mj9u
    7N0IqWQLK5DMytv7XopsZsR9QFCDNGew
  </data>
  <mac>
    3GuaasWV0XGTBbRtNP6OQ14_cHL-ZJClnaDtU6EIecw
  </mac>
</keyreq>
```

Then the sending agent performs step 5 and sends the following:

```

<iq xmlns='jabber:client'
  from='juliet@capulet.lit/balcony'
  id='xdJbWMA+'
  to='romeo@montegue.lit/garden'
  type='result'>
  <keyreq xmlns='urn:ietf:params:xml:ns:xmpp-e2e:6'
    id='835c92a8-94cd-4e96-b3f3-b2e75a438f92'>
    <encheader>
      eyJhbGciOiJSU0EtT0FFUCIsImtpZCI6InJvbWVvQG1vbnRlZ3VlLmxpdC9
      nYXJkZW4iLCJlbmMiOiJBMjU2Q0JDK0hTNTYyIiwiaWF0IjoiYXBwbGljYX
      Rpb24vandrK2pzb24ifQ
    </encheader>
    <cmk>
      hKUOpAif76c-hmRwEphVB9wXjloLpwu75x98MSWyCBtfUgmopk93ttUXoZ4
      AAiKrZJOtrPUqPZwYHjay3ggfgjVljJ_KGhgqI5cScIzaAQs0Pxep6Fnrsn
      Urw09Sjv2VRXOay4guMQnbQo0ibpifBxeuL9MJ_vdeb_BdSE8YZ4iTfMb7G
      T35gZC9NgweX3fiTEo2LjY8hEV3DHud5LlNZzYp9kLmAUZNIwGu7LtYyI4F
      7NnOv9oLx1HtmfE3_skkytQoKMvMewLkIO88h325qCpWFdrLwPp63betCme
      wDJPaBdrp9lrLchkXVo-d2ueKkb59TxWjMx7esBdaxCAcdQ
    </cmk>
    <iv>
      Ggiego8UiSsj7GgY94qOng
    </iv>
    <data>
      4vIGDz9Hm6X4lSo9JoA6ZzS0KitztLGaiMUs3RTviFO09choPhxJNlOj8KX
      8QILu4zZ-ytCnG-yzNx5SsT8KEQJhIf6_9yWplxpXl73k6ZJV-sXGd4Mj9u
      7N0IqWQLK5DMytv7XopsZsR9QFCDNGew
    </data>
    <mac>
      3GuaasWV0XGTBBrtNP6OQ14_cHL-ZJClnaDtU6EIecw
    </mac>
  </keyreq>
</iq>

```

9. Multiple Operations

The individual processes for encrypting and signing can be nested; the output of each process a complete stanza that could then be performed with the other. An implementation **MUST** be able to process one level of nesting (e.g., an encrypted stanza nested within a signed stanza), and **SHOULD** handle multiple levels within reasonable limits for the receiving agent.

10. Inclusion and Checking of Timestamps

Timestamps are included to help prevent replay attacks. All timestamps **MUST** conform to [XEP-0082] and be presented as UTC with no offset, and **SHOULD** include the seconds and fractions of a second to

three digits. Absent a local adjustment to the sending agent's perceived time or the underlying clock time, the sending agent MUST ensure that the timestamps it sends to the receiver increase monotonically (if necessary by incrementing the seconds fraction in the timestamp if the clock returns the same time for multiple requests). The following rules apply to the receiving agent:

- o It MUST verify that the timestamp received is within an acceptable range of the current time. It is RECOMMENDED that implementations use an acceptable range of five minutes, although implementations MAY use a smaller acceptable range.
- o It SHOULD verify that the timestamp received is greater than any timestamp received in the last 10 minutes which passed the previous check.
- o If any of the foregoing checks fails, the timestamp SHOULD be presented to the receiving entity (human or application) marked as "old timestamp", "future timestamp", or "decreasing timestamp", and the receiving entity MAY return a stanza error to the sender.

Note the foregoing assumes the stanza is received while the receiving agent is online; see Section 12 for offline storage considerations.

11. Interaction with Stanza Semantics

The following limitations and caveats apply:

- o Undirected <presence/> stanzas SHOULD NOT be encrypted. Such stanzas are delivered to anyone the sender has authorized, and can generate a large volume of key requests.
- o Undirected <presence/> stanzas MAY be signed. However, note that signatures significantly increase the size of a stanza kind that is often multiplexed across to many XMPP entities; this could have large impacts on bandwidth and latency.
- o Stanzas directed to multiplexing services (e.g., multi-user chat) SHOULD NOT be encrypted, unless the sender has established an acceptable trust relationship with the multiplexing service.

12. Interaction with Offline Storage

The server makes its best effort to deliver stanzas. When the receiving agent is offline at the time of delivery, the server might store the message until the recipient is next online (offline storage does not apply to <iq/> or <presence/> stanzas, only <message/> stanzas). The following need to be considered:

- o If the sending agent is not also online when the message is delivered to the receiving agent from offline storage, then the decryption process fails for insufficient information as described in Section 6.3.3.
- o When performing the timestamp checks in Section 10, if the server includes delayed delivery data as specified in [XEP-0203] for when the server received the message, then the receiving agent SHOULD use the delayed delivery timestamp rather than the current time.

13. Mandatory-to-Implement Cryptographic Algorithms

All algorithms that MUST be implemented for [JOSE-JWE] and [JOSE-JWS] also MUST be implemented for this specification. However, this specification further mandates the use of the following:

- o MUST implement the "RSA1_5" JWE algorithm.
- o MUST implement the "RS256" JWS algorithm.

14. Security Considerations

14.1. Storage of Encrypted Stanzas

The recipient's server might store any <message/> stanzas received until the recipient is next available; this duration could be anywhere from a few minutes to several months.

14.2. Re-use of Session Master Keys

A sender SHOULD NOT use the same SMK for stanzas intended for different recipients, as determined by the localpart and domainpart of the recipient's JID.

A sender MAY re-use a SMK for several stanzas to the same recipient. In this case, the SID remains the same, but the sending agent MUST

generate a new CMK and IV for each encrypted stanza. The sender SHOULD periodically generate a new SMK (and its associated SID); however, this specification does not mandate any specific algorithms or processes.

In the case of <message/> stanzas, a sending agent might generate a new SMK each time it generates a new ThreadID, as outlined in [XEP-0201].

15. IANA Considerations

15.1. XML Namespaces Name for e2e Data in XMPP

A number of URN sub-namespaces of encrypted and/or signed content for the Extensible Messaging and Presence Protocol (XMPP) is defined as follows.

URI: urn:ietf:params:xml:ns:xmpp-e2e:6

Specification: RFC XXXX

Description: This is an XML namespace name of encrypted and/or signed content for the Extensible Messaging and Presence Protocol as defined [[this document]].

Registrant Contact: IESG, <iesg@ietf.org>

URI: urn:ietf:params:xml:ns:xmpp-e2e:6:encryption

Specification: RFC XXXX

Description: This is an XML namespace name signalling support for encrypted content for the Extensible Messaging and Presence Protocol as defined [[this document]].

Registrant Contact: IESG, <iesg@ietf.org>

URI: urn:ietf:params:xml:ns:xmpp-e2e:6:signatures

Specification: RFC XXXX

Description: This is an XML namespace name signalling support for signed content for the Extensible Messaging and Presence Protocol as defined [[this document]].

Registrant Contact: IESG, <iesg@ietf.org>

16. References

16.1. Normative References

- [E2E-REQ] Saint-Andre, P., "Requirements for End-to-End Encryption in the Extensible Messaging and Presence Protocol (XMPP)", draft-saintandre-xmpp-e2e-requirements-01 (work in progress), March 2010.
- [JOSE-JWA] Jones, M., "JSON Web Algorithms (JWA)", draft-ietf-jose-json-web-algorithms-11 (work in progress), May 2013.
- [JOSE-JWE] Jones, M., Rescola, E., and J. Hildebrand, "JSON Web Encryption (JWE)", draft-ietf-jose-json-web-encryption-11 (work in progress), May 2013.
- [JOSE-JWK] Jones, M., "JSON Web Key (JWK)", draft-ietf-jose-json-web-key-11 (work in progress), December 2012.
- [JOSE-JWS] Jones, M., Bradley, J., and N. Sakimura, "JSON Web Signature (JWS)", draft-ietf-jose-json-web-signature-11 (work in progress), May 2013.
- [JOSE-KEYPROTECT] Miller, M., "Using JSON Web Encryption (JWE) for Protecting JSON Web Key (JWK) Objects", draft-miller-jose-jwe-protected-jwk-00 (work in progress), February 2013.
- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.
- [RFC4648] Josefsson, S., "The Base16, Base32, and Base64 Data Encodings", RFC 4648, October 2006.
- [RFC4949] Shirey, R., "Internet Security Glossary, Version 2", RFC 4949, August 2007.
- [RFC6120] Saint-Andre, P., "Extensible Messaging and Presence Protocol (XMPP): Core", RFC 6120, March 2011.
- [RFC6121] Saint-Andre, P., "Extensible Messaging and Presence Protocol (XMPP): Instant Messaging and Presence", RFC 6121, March 2011.

- [XEP-0030] Eatmon, R., Hildebrand, J., Millard, P., and P. Saint-Andre, "Service Discovery", XSF XEP 0030, June 2006.
- [XEP-0082] Saint-Andre, P., "XMPP Date and Time Profiles", XSF XEP 0082, May 2003.
- [XEP-0115] Hildebrand, J., Troncon, R., and P. Saint-Andre, "Entity Capabilities", XSF XEP 0115, February 2008.
- [XEP-0203] Saint-Andre, P., "Delayed Delivery", XSF XEP 0203, September 2009.
- [XEP-0297] Wild, M. and K. Smith, "Stanza Forwarding", XSF XEP 0297, July 2012.

16.2. Informative References

- [RFC3923] Saint-Andre, P., "End-to-End Signing and Object Encryption for the Extensible Messaging and Presence Protocol (XMPP)", RFC 3923, October 2004.
- [RFC4086] Eastlake, D., Schiller, J., and S. Crocker, "Randomness Requirements for Security", RFC 4086, June 2005.
- [XEP-0201] Saint-Andre, P., Paterson, I., and K. Smith, "Best Practices for Message Threads", XSF XEP 0203, November 2010.
- [Key-Table] Housley, R., Polk, T., Hartman, S., and D. Zhang, "Database of Long-Lived Symmetric Cryptographic Keys", December 2013.

Appendix A. Schema for urn:ietf:params:xml:ns:xmpp-e2e:6

The following XML schema is descriptive, not normative.

```
<?xml version='1.0' encoding='UTF-8'?>

<xs:schema
  xmlns:xs='http://www.w3.org/2001/XMLSchema'
  targetNamespace='urn:ietf:params:xml:ns:xmpp-e2e:6'
```

```
xmlns='urn:ietf:params:xml:ns:xmpp-e2e:6'
elementFormDefault='qualified'>

<xs:element name='e2e'>
  <xs:complexType>
    <xs:attribute name='id' type='xs:string' use='optional' />
    <xs:attribute name='type' use='required'>
      <xs:simpleType>
        <xs:restriction base='xs:NMTOKEN'>
          <xs:enumeration value='enc' />
          <xs:enumeration value='sig' />
        </xs:restriction>
      </xs:simpleType>
    </xs:attribute>
    <xs:sequence>
      <xs:element ref='header' minOccurs='1' maxOccurs='1' />
      <xs:element ref='cmk' minOccurs='1' maxOccurs='1' />
      <xs:element ref='iv' minOccurs='1' maxOccurs='1' />
      <xs:element ref='data' minOccurs='1' maxOccurs='1' />
      <xs:element ref='mac' minOccurs='1' maxOccurs='1' />
    </xs:sequence>
  </xs:complexType>
</xs:element>

<xs:element name='keyreq'>
  <xs:complexType>
    <xs:attribute name='id' type='xs:string' use='required' />
    <xs:sequence>
      <xs:element ref='pkey' minOccurs='0' maxOccurs='1' />
      <xs:element ref='header' minOccurs='0' maxOccurs='1' />
      <xs:element ref='cmk' minOccurs='1' maxOccurs='1' />
      <xs:element ref='iv' minOccurs='1' maxOccurs='1' />
      <xs:element ref='data' minOccurs='1' maxOccurs='1' />
      <xs:element ref='mac' minOccurs='1' maxOccurs='1' />
    </xs:sequence>
  </xs:complexType>
</xs:element>

<xs:element name='cmk'>
  <xs:complexType>
    <xs:simpleType>
      <xs:extension base='xs:string'>
      </xs:extension>
    </xs:simpleType>
  </xs:complexType>
</xs:element>

<xs:element name='iv'>
```

```
<xs:complexType>
  <xs:simpleType>
    <xs:extension base='xs:string'>
    </xs:extension>
  </xs:simpleType>
</xs:complexType>
</xs:element>

<xs:element name='data'>
  <xs:complexType>
    <xs:simpleType>
      <xs:extension base='xs:string'>
      </xs:extension>
    </xs:simpleType>
  </xs:complexType>
</xs:element>

<xs:element name='encheader'>
  <xs:complexType>
    <xs:simpleType>
      <xs:extension base='xs:string'>
      </xs:extension>
    </xs:simpleType>
  </xs:complexType>
</xs:element>

<xs:element name='mac'>
  <xs:complexType>
    <xs:simpleType>
      <xs:extension base='xs:string'>
      </xs:extension>
    </xs:simpleType>
  </xs:complexType>
</xs:element>

<xs:element name='pkey'>
  <xs:complexType>
    <xs:simpleType>
      <xs:extension base='xs:string'>
      </xs:extension>
    </xs:simpleType>
  </xs:complexType>
</xs:element>

<xs:element name='sigheader'>
  <xs:complexType>
    <xs:simpleType>
      <xs:extension base='xs:string'>
```

```
        </xs:extension>
      </xs:simpleType>
    </xs:complexType>
  </xs:element>

  <xs:element name='bad-timestamp' type='empty' />
  <xs:element name='decryption-failed' type='empty' />
  <xs:element name='insufficient-information' type='empty' />
  <xs:element name='verification-failed' type='empty' />

  <xs:simpleType name='empty'>
    <xs:restriction base='xs:string'>
      <xs:enumeration value='' />
    </xs:restriction>
  </xs:simpleType>

</xs:schema>
```

Appendix B. Acknowledgements

Thanks to Richard Barnes, Andrew Biggs, and Ben Schumacher for their feedback.

Authors' Addresses

Matthew Miller
Cisco Systems, Inc.
1899 Wynkoop Street, Suite 600
Denver, CO 80202
USA

Phone: +1-303-308-3204
Email: mamille2@cisco.com

Carl Wallace
Red Hound Software, Inc.

Email: carl@redhoundsoftware.com