Internet Engineering Task Force (IETF)
Request for Comments: 7516
Category: Standards Track
ISSN: 2070-1721

M. Jones
Microsoft
J. Hildebrand
Cisco

May 2015

# JSON Web Encryption (JWE)

## Abstract

JSON Web Encryption (JWE) represents encrypted content using 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 Message Authentication Code (MAC) capabilities are described in the separate JSON Web Signature (JWS) specification.

## Status of This Memo

This is an Internet Standards Track document.

This document is a product of the Internet Engineering Task Force (IETF). It represents the consensus of the IETF community. It has received public review and has been approved for publication by the Internet Engineering Steering Group (IESG). Further information on Internet Standards is available in Section 2 of RFC 5741.

Information about the current status of this document, any errata, and how to provide feedback on it may be obtained at http://www.rfc-editor.org/info/rfc7516.

## 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

JSON Web Encryption (JWE) represents encrypted content using JSON-based data structures [RFC7159]. 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]. The interpretation should only be applied when the terms appear in all capital letters.

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

The terms "JSON Web Signature (JWS)", "Base64url Encoding", "Collision-Resistant Name", "Header Parameter", "JOSE Header", and "StringOrURI" are defined by the JWS specification [JWS].

The terms "Ciphertext", "Digital Signature", "Initialization Vector (IV)", "Message Authentication Code (MAC)", and "Plaintext" are defined by the "Internet Security Glossary, Version 2" [RFC4949].

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 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 the JWE Compact Serialization or the JWE JSON 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 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 CEK value is encrypted to the intended recipient using an asymmetric encryption algorithm.

Key Wrapping

A Key Management Mode in which the 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 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 CEK value to the intended recipient using a symmetric key wrapping algorithm.

Direct Encryption

A Key Management Mode in which the 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 whitespace 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):

- JOSE Header
- JWE Encrypted Key
- JWE Initialization Vector
- JWE AAD
- JWE Ciphertext
- 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):

- JWE Protected Header
- JWE Shared Unprotected Header
- 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:

- The Content Encryption Key is encrypted to the recipient using the RSAES-OAEP [RFC3447] algorithm to produce the JWE Encrypted Key.

- 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:

```
eyJhbGciOiJSU0EtT0FFUCIsImVuYyI6IkEyNTZHQ00ifQ
```

The remaining steps to finish creating this JWE are:

- Generate a random Content Encryption Key (CEK).

- Encrypt the CEK with the recipient’s public key using the RSAES-OAEP algorithm to produce the JWE Encrypted Key.

- Base64url-encode the JWE Encrypted Key.

- Generate a random JWE Initialization Vector.

- Base64url-encode the JWE Initialization Vector.

- Let the Additional Authenticated Data encryption parameter be ASCII(BASE64URL(UTF8(JWE Protected Header))).

- 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.

- Base64url-encode the ciphertext.

- Base64url-encode the Authentication Tag.

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

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

```
eyJhbGciOiJSU0EtT0FFUCIsImVuYyI6IkEyNTZHQ00ifQ.
OKOawDo13gRp2ojaHV7LFpZcgV7T6DVZKTyKOMTYUmKoTCVJRgckCL9kiMT03JGe
ipsEdY3mx_etLbbWSrFr05kLzcSr4qKAq7YN7e9jwQRb23nfa6c9d-StnImGyFDb
Sv04uVuxIp5Zms1gNxKKK2Da14B8S4rzVRltdYwam_lDp5XnZAYpQdb76FdIKLaV
mqgfwX7XWRxv2322i-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 established by [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 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 established by [JWA]; the initial contents of this registry are the values defined in Section 4.1 of [JWA].

### 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 established by [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 established by [JWA]; the initial contents of this registry are the values defined in Section 5.1 of [JWA].

### 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 Algorithms" registry established by [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 established by [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 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 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 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 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: 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 fail, 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, whitespace, 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 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 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 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 an empty octet sequence.

11. When Direct Encryption is employed, let the 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 Transport Layer Security (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.

- 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.

- 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.

- 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.)

- 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 section registers the Header Parameter names defined in Section 4.1 in the IANA "JSON Web Signature and Encryption Header Parameters" registry established by [JWS].

#### 10.1.1. Registry Contents

- Header Parameter Name: "alg"
- Header Parameter Description: Algorithm
- Header Parameter Usage Location(s): JWE
- Change Controller: IESG
- Specification Document(s): Section 4.1.1 of RFC 7516

- Header Parameter Name: "enc"
- Header Parameter Description: Encryption Algorithm
- Header Parameter Usage Location(s): JWE
- Change Controller: IESG
- Specification Document(s): Section 4.1.2 of RFC 7516

- Header Parameter Name: "zip"
- Header Parameter Description: Compression Algorithm
- Header Parameter Usage Location(s): JWE
- Change Controller: IESG
- Specification Document(s): Section 4.1.3 of RFC 7516

- Header Parameter Name: "jku"
- Header Parameter Description: JWK Set URL
- Header Parameter Usage Location(s): JWE
- Change Controller: IESG
- Specification Document(s): Section 4.1.4 of RFC 7516

- Header Parameter Name: "jwk"
- Header Parameter Description: JSON Web Key
- Header Parameter Usage Location(s): JWE

- Change Controller: IESG
- Specification Document(s): Section 4.1.5 of RFC 7516

- Header Parameter Name: "kid"
- Header Parameter Description: Key ID
- Header Parameter Usage Location(s): JWE
- Change Controller: IESG
- Specification Document(s): Section 4.1.6 of RFC 7516

- Header Parameter Name: "x5u"
- Header Parameter Description: X.509 URL
- Header Parameter Usage Location(s): JWE
- Change Controller: IESG
- Specification Document(s): Section 4.1.7 of RFC 7516

- Header Parameter Name: "x5c"
- Header Parameter Description: X.509 Certificate Chain
- Header Parameter Usage Location(s): JWE
- Change Controller: IESG
- Specification Document(s): Section 4.1.8 of RFC 7516

- Header Parameter Name: "x5t"
- Header Parameter Description: X.509 Certificate SHA-1 Thumbprint
- Header Parameter Usage Location(s): JWE
- Change Controller: IESG
- Specification Document(s): Section 4.1.9 of RFC 7516

- Header Parameter Name: "x5t#S256"
- Header Parameter Description: X.509 Certificate SHA-256 Thumbprint
- Header Parameter Usage Location(s): JWE
- Change Controller: IESG
- Specification Document(s): Section 4.1.10 of RFC 7516

- Header Parameter Name: "typ"
- Header Parameter Description: Type
- Header Parameter Usage Location(s): JWE
- Change Controller: IESG
- Specification Document(s): Section 4.1.11 of RFC 7516

- Header Parameter Name: "cty"
- Header Parameter Description: Content Type
- Header Parameter Usage Location(s): JWE
- Change Controller: IESG
- Specification Document(s): Section 4.1.12 of RFC 7516

- Header Parameter Name: "crit"
- Header Parameter Description: Critical
- Header Parameter Usage Location(s): JWE

- Change Controller: IESG
- Specification Document(s): Section 4.1.13 of RFC 7516

## 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" Header 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)", RFC 7518, DOI 10.17487/RFC7518, May 2015, <http://www.rfc-editor.org/info/rfc7518>.

[JWK] Jones, M., "JSON Web Key (JWK)", RFC 7517, DOI 10.17487/RFC7517, May 2015, <http://www.rfc-editor.org/info/rfc7517>.

[JWS] Jones, M., Bradley, J., and N. Sakimura, "JSON Web Signature (JWS)", RFC 7515, DOI 10.17487/RFC7515, May 2015, <http://www.rfc-editor.org/info/rfc7515>.

[RFC1951] Deutsch, P., "DEFLATE Compressed Data Format Specification version 1.3", RFC 1951, DOI 10.17487/RFC1951, May 1996, <http://www.rfc-editor.org/info/rfc1951>.

[RFC20] Cerf, V., "ASCII format for Network Interchange", STD 80, RFC 20, DOI 10.17487/RFC0020, October 1969, <http://www.rfc-editor.org/info/rfc20>.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <http://www.rfc-editor.org/info/rfc2119>.

[RFC3629] Yergeau, F., "UTF-8, a transformation format of ISO 10646", STD 63, RFC 3629, DOI 10.17487/RFC3629, November 2003, <http://www.rfc-editor.org/info/rfc3629>.

[RFC4949] Shirey, R., "Internet Security Glossary, Version 2", FYI 36, RFC 4949, DOI 10.17487/RFC4949, August 2007, <http://www.rfc-editor.org/info/rfc4949>.

[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, DOI 10.17487/RFC5280, May 2008, <http://www.rfc-editor.org/info/rfc5280>.

[RFC7159] Bray, T., Ed., "The JavaScript Object Notation (JSON) Data Interchange Format", RFC 7159, DOI 10.17487/RFC7159, March 2014, <http://www.rfc-editor.org/info/rfc7159>.

[UNICODE] The Unicode Consortium, "The Unicode Standard", <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, <http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf>.

[JSE] Bradley, J. and N. Sakimura (editor), "JSON Simple Encryption", September 2010, <http://jsonenc.info/enc/1.0/>.

[JSMS] Rescorla, E. and J. Hildebrand, "JavaScript Message Security Format", Work in Progress, draft-rescorla-jsms-00, March 2011.

[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, November 2007, <http://csrc.nist.gov/publications/nistpubs/800-38D/SP-800-38D.pdf>.

[RFC3218] Rescorla, E., "Preventing the Million Message Attack on Cryptographic Message Syntax", RFC 3218, DOI 10.17487/RFC3218, January 2002, <http://www.rfc-editor.org/info/rfc3218>.

[RFC3447] Jonsson, J. and B. Kaliski, "Public-Key Cryptography Standards (PKCS) #1: RSA Cryptography Specifications Version 2.1", RFC 3447, DOI 10.17487/RFC3447, February 2003, <http://www.rfc-editor.org/info/rfc3447>.

[RFC3766] Orman, H. and P. Hoffman, "Determining Strengths For Public Keys Used For Exchanging Symmetric Keys", BCP 86, RFC 3766, DOI 10.17487/RFC3766, April 2004, <http://www.rfc-editor.org/info/rfc3766>.

[RFC4086] Eastlake 3rd, D., Schiller, J., and S. Crocker, "Randomness Requirements for Security", BCP 106, RFC 4086, DOI 10.17487/RFC4086, June 2005, <http://www.rfc-editor.org/info/rfc4086>.

[RFC5652] Housley, R., "Cryptographic Message Syntax (CMS)", STD 70, RFC 5652, DOI 10.17487/RFC5652, September 2009, <http://www.rfc-editor.org/info/rfc5652>.

[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:

- The Content Encryption Key is encrypted to the recipient using the RSAES-OAEP algorithm to produce the JWE Encrypted Key.
- 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 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
      cJoZmds2h7M70imEVhRU5djINXtqllXI4DFqcI1DgjT9LewND8MW2Krf3S
      psk_ZkoFnilakGygTwpZ3uesH-PFABNIUYpOiN15dsQRkgr0vEhxN92i2a
      sbOenSZeyaxziK72UwxrrKoExv6kc5twXTq4h-QChLOln0_mtUZwfsRaMS
      tPs6mS6XrgxnxbWhojf663tuEQueGC-FCMfra36C9knDFGzKsNa7LZK2dj
      YgyD3JR_MB_4NUJW_TqOQtwHYbxevoJArm-L5StowjzGy-_bq6Gw",
 "e":"AQAB",
 "d":"kLdtIj6GbDks_ApCSTYQtelcNttlKiOyPzMrXHeI-yk1F7-kpDxY4-WY5N
      WV5KntaEeXS1j82E375xxhWMHXyvjYecPT9fpwR_M9gV8n9Hrh2anTpTD9
      3Dt62ypW3yDsJzBnTnrYu1iwWRgBKrEYY46qAZIrA2xAwnm2X7uGR1hghk
      qDp0Vqj3kbSCz1XyfCs6_LehBwtxHIyh8Ripy40p24moOAbgxVw3rxT_vl
      t3UVe4WO3JkJOzlpUf-KTVI2Ptgm-dARxTEtE-id-4OJr0h-K-VFs3VSnd
      VTIznSxfyrj8ILL6MG_Uv8YAu7VILSB3lOW085-4qE3DzgrTjgyQ",
 "p":"1r52Xk46c-LsfB5P442p7atdPUrxQSy4mti_tZI3Mgf2EuFVbUoDBvaRQ-
      SWxkbkmoEzL7JXroSBjSrK3YIQgYdMgyAEPTPjXv_hI2_1eTSPVZfzL0lf
      fNn03IXqWF5MDFuoUYE0hzb2vhrlN_rKrbfDIwUbTrjjgieRbwC6Cl0",
 "q":"wLb35x7hmQWZsWJmB_vle87ihgZ19S8lBEROLIsZG4ayZVe9Hi9gDVCOBm
      UDdaDYVTSNx_8Fyw1YYa9XGrGnDew00J28cRUoeBB_jKI1oma0Orv1T9aX
      IWxKwd4gvxFImOWr3QRL9KEBRzk2RatUBnmDZJTIAfwTs0g68UZHvtc",
 "dp":"ZK-YwE7diUh0qR1tR7w8WHtolDx3MZ_OTowiFvgfeQ3SiresXjm9gZ5KL
      hMXvo-uz-KUJWDxS5pFQ_M0evdo1dKiRTjVw_x4NyqyXPM5nULPkcpU827
      rnpZzAJKpdhWAgqrXGKAECQH0Xt4taznjnd_zVpAmZZq60WPMBMfKcuE",
 "dq":"Dq0gfgJ1DdFGXiLvQEZnuKEN0UUmsJBxkjydc3j4ZYdBiMRAy86x0vHCj
      ywcMlYYg4yoC4YZa9hNVcsjqA3FeiL19rk8g6Qn29Tt0cj8qqyFpz9vNDB
      UfCAiJVeESOjJDZPYHdHY8v1b-o-Z2X5tvLx-TCekf7oxyeKDUqKWjis",
 "qi":"VIMpMYbPf47dT1w_zDUXfPimsSegnMOA1zTaX7aGk_8urY6R8-ZW1FxU7
      AlWAyLWybqq6t16VFd7hQd0y6flUK4SlOydB61gwanOsXGOAOv82cHq0E3
      eL4HrtZkUuKvnPrMnsUUFlfUdybVzxyjz9JF_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
Sv04uVuxIp5Zms1gNxKKK2Da14B8S4rzVRltdYwam_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:

```
48V1_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:

```
  eyJhbGciOiJSU0EtT0FFUCIsImVuYyI6IkEyNTZHQ00ifQ.
  OKOawDo13gRp2ojaHV7LFpZcgV7T6DVZKTyKOMTYUmKoTCVJRgckCL9kiMT03JGe
  ipsEdY3mx_etLbbWSrFr05kLzcSr4qKAq7YN7e9jwQRb23nfa6c9d-StnImGyFDb
  Sv04uVuxIp5Zms1gNxKKK2Da14B8S4rzVRltdYwam_lDp5XnZAYpQdb76FdIKLaV
  mqgfwX7XWRxv2322i-vDxRfqNzo_tETKzpVLzfiwQyeyPGLBIO56YJ7eObdv0je8
  1860ppamavo35UgoRdbYaBcoh9QcfylQr66oc6vFWXRcZ_ZT2LawVCWTIy3brGPi
  6UklfCpIMfIjf7iGdXKHzg.
  48V1_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:

- The Content Encryption Key is encrypted to the recipient using the RSAES-PKCS1-v1_5 algorithm to produce the JWE Encrypted Key.
- 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 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
        UzewbgBHod5pcM9H95GQRV3JDXboIRROSBigeC5yjU1hGzHHyXss8UDpre
        cbAYxknTcQkhslANGRUZmdTOQ5qTRsLAt6BTYuyvVRdhS8exSZEy_c4gs_
        7svlJJQ4H9_NxsiIoLwAEk7-Q3UXERGYw_75IDrGA84-lA_-Ct4eTlXHBI
        Y2EaV7t7LjJaynVJCpkv4LKjTTAumiGUIuQhrNhZLuF_RJLqHpM2kgWFLU
        7-VTdL1VbC2tejvcI2BlMkEpk1BzBZI0KQB0GaDWFLN-aEAw3vRw",
   "e":"AQAB",
   "d":"VFCWOqXr8nvZNyaaJLXdnNPXZKRaWCjkU5Q2egQQpTBMwhprMzWzpR8Sxq
        1OPThh_J6MUD8Z35wky9b8eEO0pwNS8xlh1lOFRRBoNqDIKVOku0aZb-ry
        nq8cxjDTLZQ6Fz7jSjR1Klop-YKaUHc9GsEofQqYruPhzSA-QgajZGPbE_
        0ZaVDJHfyd7UUBUKunFMScbflYAAOYJqVIVwaYR5zWEEceUjNnTNo_CVSj
        -VvXLO5VZfCUAVLgW4dpf1SrtZjSt34YLsRarSb127reG_DUwg9Ch-Kyvj
        T1SkHgUWRVGcyly7uvVGRSDwsXypdrNinPA4jlhoNdizK2zF2CWQ",
   "p":"9gY2w6I6S6L0juEKsbeDAwpd9WMfgqFoeA9vEyEUuk4kLwBKcoe1x4HG68
        ik918hdDSE9vDQSccA3xXHOAFOPJ8R9EeIAbTi1VwBYnbTp87X-xcPWlEP
        krdoUKW60tgs1aNd_Nnc9LEVVPMS390zbFxt8TN_biaBgelNgbC95sM",
   "q":"uKlCKvKv_ZJMVcdIs5vVSU_6cPtYI1ljWytExV_skstvRSNi9r66jdd9-y
        BhVfuG4shsp2j7rGnIio901RBeHo6TPKWVVykPu1iYhQXw1jIABfw-MVsN
        -3bQ76WLdt2SDxsHs7q7zPyUyHXmps7ycZ5c72wGkUwNOjYelmkiNS0",
   "dp":"w0kZbV63cVRvVX6yk3C8cMxo2qCM4Y8nsq1lmMSYhG4EcL6FWbX5h9yuv
        ngs4iLEFk6eALoUS4vIWEwcL4txw9LsWH_zKI-hwoReoP77cOdSL4AVcra
        Hawlkpyd2TWjE5evgbhWtOxnZee3cXJBkAi64Ik6jZxbvk-RR3pEhnCs",
   "dq":"o_8V14SezckO6CNLKs_btPdFiO9_kC1DsuUTd2LAfIIVeMZ7jn1Gus_Ff
        7B7IVx3p5KuBGOVF8L-qifLb6nQnLysgHDh132NDioZkhH7mI7hPG-PYE_
        odApKdnqECHWw0J-F0JWnUd6D2B_1TvF9mXA2Qx-iGYn8OVV1Bsmp6qU",
   "qi":"eNho5yRBEBxhGBtQRww9QirZsB66TrfFReG_CcteI1aCneT0ELGhYlRlC
        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_NPVaXsGMoLOmwvc1GyqlIKOK1nN94nHPoltGRhWhw7Zx0-kFm
  1NJn8LE9XShH59_i8J0PH5ZZyNfGy2xGdULU7sHNF6Gp2vPLgNZ__deLKxGHZ7Pc
  HALUzoOegEI-8E66jX2E4zyJKx-YxzZIItRzC5hlRirb6Y5Cl_p-ko3YvkkysZIF
  NPccxRU7qve1WYPxqbb2Yw8kZqa2rMWI5ng8OtvzlV7elprCbuPhcCdZ6XDP0_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_NPVaXsGMoLOmwvc1GyqlIKOK1nN94nHPoltGRhWhw7Zx0-kFm
1NJn8LE9XShH59_i8J0PH5ZZyNfGy2xGdULU7sHNF6Gp2vPLgNZ__deLKxGHZ7Pc
HALUzoOegEI-8E66jX2E4zyJKx-YxzZIItRzC5hlRirb6Y5Cl_p-ko3YvkkysZIF
NPccxRU7qve1WYPxqbb2Yw8kZqa2rMWI5ng8OtvzlV7elprCbuPhcCdZ6XDP0_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:

- 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.
- 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:

```
eyJhbGciOiJBMTI4S1ciLCJlbmMiOiJBMTI4Q0JDLUhTMjU2In0
```

### A.3.2. Content Encryption Key (CEK)

Generate a 256-bit random CEK. In this example, the value is:

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

### A.3.3. Key Encryption

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

```
{"kty":"oct",
 "k":"GawgguFyGrWKav7AX4VKUg"
}
```

The resulting JWE Encrypted Key value is:

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

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

```
6KB707dM9YTIgHtLvtgWQ8mKwboJW3of9locizkDTHzBC2IlrT1oOQ
```

### 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.
6KB707dM9YTIgHtLvtgWQ8mKwboJW3of9locizkDTHzBC2IlrT1oOQ.
AxY8DCtDaGlsbGljb3RoZQ.
KDlTtXchhZTGufMYmOYGS4HffxPSUrfmqCHXaI9wOGY.
U0m_YmjN04DJvceFICbCVQ
```

### A.3.8. Validation

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

## A.4. Example JWE Using 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 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 CEK. The second uses AES Key Wrap to encrypt the CEK. Key ID values are supplied for both keys. The two JWE Per-Recipient Unprotected 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 Shared Unprotected Header

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

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

### A.4.4. Complete JOSE Header Values

Combining the JWE Per-Recipient Unprotected Header, JWE Protected Header, and JWE Shared 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_NPVaXsGMoLOmwvc1GyqlIKOK1nN94nHPoltGRhWhw7Zx0-
       kFm1NJn8LE9XShH59_i8J0PH5ZZyNfGy2xGdULU7sHNF6Gp2vPLgNZ__deLKx
       GHZ7PcHALUzoOegEI-8E66jX2E4zyJKx-YxzZIItRzC5hlRirb6Y5Cl_p-ko3
       YvkkysZIFNPccxRU7qve1WYPxqbb2Yw8kZqa2rMWI5ng8OtvzlV7elprCbuPh
       cCdZ6XDP0_F8rkXds2vE4X-ncOIM8hAYHHi29NX0mcKiRaD0-D-ljQTP-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 Public-Key Cryptography Standards (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 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]
```

## Acknowledgements

Solutions for encrypting JSON content were also explored by "JSON Simple Encryption" [JSE] and "JavaScript Message Security Format" [JSMS], both of which significantly influenced this document. This document 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 [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.

# Authors’ Addresses

Michael B. Jones
Microsoft

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

Joe Hildebrand
Cisco Systems, Inc.

EMail: jhildebr@cisco.com