JOSE Working Group
Internet-Draft
Intended status: Informational
Expires: June 27, 2015

M. Miller
Cisco Systems, Inc.

December 24, 2014

Examples of Protecting Content using JavaScript Object Signing and Encryption (JOSE)
draft-ietf-jose-cookbook-08

Abstract

This document contains a set of examples using JavaScript Object Signing and Encryption (JOSE) technology to protect data. These examples present a representative sampling JSON Web Key (JWK) objects, as well as various JSON Web Signature (JWS) and JSON Web Encryption (JWE) results given similar inputs.

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 27, 2015.

Copyright Notice

Table of Contents

1. Introduction

The JavaScript Object Signing and Encryption (JOSE) technologies - JSON Web Signature (JWS) [I-D.ietf-jose-json-web-signature], JSON Web Encryption (JWE) [I-D.ietf-jose-json-web-encryption], JSON Web Key (JWK) [I-D.ietf-jose-json-web-key], and JSON Web Algorithms (JWA) [I-D.ietf-jose-json-web-algorithms] - collectively can be used to encrypt and/or sign content using a variety of algorithms. While the full set of permutations is extremely large, and might be daunting to some, it is expected that most applications will only use a small set of algorithms to meet their needs.

This document provides a number of examples of signing or encrypting content using JOSE. While not exhaustive, it does compile a representative sample of JOSE features. As much as possible, the same signature payload or encryption plaintext content is used to illustrate differences in various signing and encryption results.

This document also provides a number of example JWK objects. These examples illustrate the distinguishing properties of various key types, and emphasize important characteristics. Most of the JWK examples are then used in the signature or encryption examples that follow.

All of the examples contained herein are available in a machine-readable format at https://github.com/ietf-jose/cookbook.

1.1. Conventions Used in this Document

This document separates data that are expected to be input to an implementation of JOSE from data that are expected to be generated by an implementation of JOSE. Each example, wherever possible, provides enough information to both replicate the results of this document or to validate the results by running its inverse operation (e.g., signature results can be validated by performing the JWS verify). However, some algorithms inherently use random data and therefore computations employing them cannot be exactly replicated; such cases are explicitly stated in the relevant sections.

All instances of binary octet strings are represented using [RFC4648] base64url encoding.

Wherever possible and unless otherwise noted, the examples include the Compact serialization, JSON General Serialization, and JSON Flattened Serialization.

All of the examples in this document have whitespace added to improve formatting and readability. Except for JWE plaintext or JWS payload content, whitespace is not part of the cryptographic operations nor the exchange results.

Unless otherwise noted, the JWE plaintext or JWS payload content does include " " (U+0020 SPACE) characters. Line breaks (U+000A LINE FEED) replace some " " (U+0020 SPACE) characters to improve readability but are not present in the JWE plaintext or JWS payload.

2. Terminology

This document inherits terminology regarding JSON Web Signature (JWS) technology from [I-D.ietf-jose-json-web-signature], terminology regarding JSON Web Encryption (JWE) technology from [I-D.ietf-jose-json-web-encryption], terminology regarding JSON Web Key (JWK) technology from [I-D.ietf-jose-json-web-key], and terminology regarding algorithms from [I-D.ietf-jose-json-web-algorithms].

3. JSON Web Key Examples

The following sections demonstrate how to represent various JWK and JWK-set objects.

3.1. EC Public Key

This example illustrates an Elliptic Curve public key. This example is the public key corresponding to Figure 2.

Note that whitespace is added for readability as described in Section 1.1.

```
{
  "kty": "EC",
  "kid": "bilbo.baggins@hobbiton.example",
  "use": "sig",
  "crv": "P-521",
  "x": "AHKZLLOsCOzz5cY97ewNUajB957y-C-U88c3v13nmGZx6sYl_oJXu9
      A5RkTKqjqvjyekWF-7ytDyRXYgCF5cj0Kt",
  "y": "AdymlHvOiLxXkEhayXQnNCvDX4h9htZaCJN34kfmC6pV5OhQHiraVy
      SsUdaQkAgDPrwQrJmbnX9cwlGfP-HqHZR1"
}
```

Figure 1: Elliptic Curve P-521 Public Key

The field "kty" value of "EC" identifies this as an elliptic curve key. The field "crv" identifies the curve, which is curve P-521 for this example. The fields "x" and "y" values are the base64url-encoded X and Y coordinates (respectively).

The values of the fields "x" and "y" decoded are the octets necessary to represent each full coordinate to the order of the curve. For a key over curve P-521, the values of the fields "x" and "y" are exactly 66 octets in length when decoded, padded with leading zero (0x00) octets to reach the expected length.

3.2. EC Private Key

This example illustrates an Elliptic Curve private key. This example is the private key corresponding to Figure 1.

Note that whitespace is added for readability as described in Section 1.1.

```
{
  "kty": "EC",
  "kid": "bilbo.baggins@hobbiton.example",
  "use": "sig",
  "crv": "P-521",
  "x": "AHKZLLOsCOzz5cY97ewNUajB957y-C-U88c3v13nmGZx6sYl_oJXu9
      A5RkTKqjqvjyekWF-7ytDyRXYgCF5cj0Kt",
  "y": "AdymlHvOiLxXkEhayXQnNCvDX4h9htZaCJN34kfmC6pV5OhQHiraVy
      SsUdaQkAgDPrwQrJmbnX9cwlGfP-HqHZR1",
  "d": "AAhRON2r9cqXX1hg-RoI6R1tX5p2rUAYdmpHZoC1XNM56KtscrX6zb
      KipQrCW9CGZH3T4ubpnoTKLDYJ_fF3_rJt"
}
```

Figure 2: Elliptic Curve P-521 Private Key

The field "kty" value of "EC" identifies this as an elliptic curve key. The field "crv" identifies the curve, which is curve P-521 (also known as SECG curve secp521r1) for this example. The fields "x" and "y" values are the base64url-encoded X and Y coordinates (respectively). The field "d" value is the base64url-encoded private key.

The values of the fields "d", "x", and "y" decoded are the octets necessary to represent the private key or each full coordinate (respectively) to the order of the curve. For a key over curve "P-521", the values of the "d", "x", and "y" fields are each exactly 66 octets in length when decoded, padded with leading zero (0x00) octets to reach the expected length.

3.3. RSA Public Key

This example illustrates an RSA public key. This example is the public key corresponding to Figure 4.

Note that whitespace is added for readability as described in Section 1.1.

```
{
  "kty": "RSA",
  "kid": "bilbo.baggins@hobbiton.example",
  "use": "sig",
  "n": "n4EPtAOCc9AlkeQHPzHStgAbgs7bTZLwUBZdR8_KuKPEHLd4rHVTeT
      -O-XV2jRojdNhxJWTDvNd7nqQ0VEiZQHz_AJmSCpMaJMRBSFKrKb2wqV
      wGU_NsYOYL-QtiWN2lbzcEe6XC0dApr5ydQLrHqkHHig3RBordaZ6Aj-
      oBHqFEHYpPe7Tpe-OfVfHd1E6cS6M1FZcD1NNLYD5lFHpPI9bTwJlsde
      3uhGqC0ZCuEHg8lhzwOHrtIQbS0FVbb9k3-tVTU4fg_3L_vniUFAKwuC
      LqKnS2BYwdq_mzSnbLY7h_qixoR7jig3__kRhuaxwUkRz5iaiQkqgc5g
      HdrNP5zw",
  "e": "AQAB"
}
```

Figure 3: RSA 2048-bit Public Key

The field "kty" value of "RSA" identifies this as a RSA key. The fields "n" and "e" values are the modulus and (public) exponent (respectively) using the minimum octets necessary.

For a 2048-bit key, the field "n" value is 256 octets in length when decoded.

3.4. RSA Private Key

This example illustrates an RSA private key. This example is the private key corresponding to Figure 3.

Note that whitespace is added for readability as described in Section 1.1.

```
{
  "kty": "RSA",
  "kid": "bilbo.baggins@hobbiton.example",
  "use": "sig",
  "n": "n4EPtAOCc9AlkeQHPzHStgAbgs7bTZLwUBZdR8_KuKPEHLd4rHVTeT
      -O-XV2jRojdNhxJWTDvNd7nqQ0VEiZQHz_AJmSCpMaJMRBSFKrKb2wqV
      wGU_NsYOYL-QtiWN2lbzcEe6XC0dApr5ydQLrHqkHHig3RBordaZ6Aj-
      oBHqFEHYpPe7Tpe-OfVfHd1E6cS6M1FZcD1NNLYD5lFHpPI9bTwJlsde
      3uhGqC0ZCuEHg8lhzwOHrtIQbS0FVbb9k3-tVTU4fg_3L_vniUFAKwuC
      LqKnS2BYwdq_mzSnbLY7h_qixoR7jig3__kRhuaxwUkRz5iaiQkqgc5g
      HdrNP5zw",
  "e": "AQAB",
  "d": "bWUC9B-EFRIo8kpGfh0ZuyGPvMNKvYWNtB_ikiH9k20eT-O1q_I78e
      iZkpXxXQ0UTEs2LsNRS-8uJbvQ-A1irkwMSMkK1J3XTGgdrhCku9gRld
      Y7sNA_AKZGh-Q661_42rINLRCe8W-nZ34ui_qOfkLnK9QWDDqpaIsA-b
      MwWWSDFu2MUBYwkHTMEzLYGqOe04noqeq1hExBTHBOBdkMXiuFhUq1BU
      6l-DqEiWxqg82sXt2h-LMnT3046AOYJoRioz75tSUQfGCshWTBnP5uDj
      d18kKhyv07lhfSJdrPdM5Plyl21hsFf4L_mHCuoFau7gdsPfHPxxjVOc
      OpBrQzwQ",
  "p": "3Slxg_DwTXJcb6095RoXygQCAZ5RnAvZlno1yhHtnUex_fp7AZ_9nR
      aO7HX_-SFfGQeutao2TDjDAWU4Vupk8rw9JR0AzZ0N2fvuIAmr_WCsmG
      peNqQnev1T7IyEsnh8UMt-n5CafhkikzhEsrmndH6LxOrvRJlsPp6Zv8
      bUq0k",
  "q": "uKE2dh-cTf6ERF4k4e_jy78GfPYUIaUyoSSJuBzp3Cubk3OCqs6grT
      8bR_cu0Dm1MZwWmtdqDyI95HrUeq3MP15vMMON8lHTeZu2lmKvwqW7an
      V5UzhM1iZ7z4yMkuUwFWoBvyY898EXvRD-hdqRxHlSqAZ192zB3pVFJ0
      s7pFc",
  "dp": "B8PVvXkvJrj2L-GYQ7v3y9r6Kw5g9SahXBwsWUzp19TVlgI-YV85q
      1NIb1rxQtD-IsXXR3-TanevuRPRt5OBOdiMGQp8pbt26gljYfKU_E9xn
      -RULHz0-ed9E9gXLKD4VGngpz-PfQ_q29pk5xWHoJp009Qf1HvChixRX
      59ehik",
  "dq": "CLDmDGduhylc9o7r84rEUVn7pzQ6PF83Y-iBZx5NT-TpnOZKF1pEr
      AMVeKzFEl41DlHHqqBLSM0W1sOFbwTxYWZDm6sI6og5iTbwQGIC3gnJK
      bi_7k_vJgGHwHxgPaX2PnvP-zyEkDERuf-ry4c_Z11Cq9AqC2yeL6kdK
      T1cYF8",
  "qi": "3PiqvXQN0zwMeE-sBvZgi289XP9XCQF3VWqPzMKnIgQp7_Tugo6-N
      ZBKCQsMf3HaEGBjTVJs_jcK8-TRXvaKe-7ZMaQj8VfBdYkssbu0NKDDh
      jJ-GtiseaDVWt7dcH0cfwxgFUHpQh7FoCrjFJ6h6ZEpMF6xmujs4qMpP
      z8aaI4"
}
```


Figure 4: RSA 2048-bit Private Key

The field "kty" value of "RSA" identifies this as a RSA key. The fields "n" and "e" values are the base64url-encoded modulus and (public) exponent (respectively) using the minimum number of octets necessary. The field "d" value is the base64url-encoded private exponent using the minimum number of octets necessary. The fields

"p", "q", "dp", "dq", and "qi" are the base64url-encoded additional private information using the minimum number of octets necessary.

For a 2048-bit key, the field "n" is 256 octets in length when decoded and the field "d" is not longer than 256 octets in length when decoded.

3.5. Symmetric Key (MAC Computation)

This example illustrates a symmetric key used for computing MACs.

Note that whitespace is added for readability as described in Section 1.1.

```
{
  "kty": "oct",
  "kid": "018c0ae5-4d9b-471b-bfd6-eef314bc7037",
  "use": "sig",
  "alg": "HS256",
  "k": "hJtXIZ2uSN5kbQfbtTNWbpdmhkV8FJG-Onbc6mxCcYg"
}
```

Figure 5: AES 256-bit symmetric signing key

The field "kty" value of "oct" identifies this as a symmetric key. The field "k" value is the symmetric key.

When used for the signing algorithm "HS256" (HMAC-SHA256), the field "k" value is 32 octets (or more) in length when decoded, padded with leading zero (0x00) octets to reach the minimum expected length.

3.6. Symmetric Key (Encryption)

This example illustrates a symmetric key used for encryption.

Note that whitespace is added for readability as described in Section 1.1.

```
{
  "kty": "oct",
  "kid": "1e571774-2e08-40da-8308-e8d68773842d",
  "use": "enc",
  "alg": "A256GCM",
  "k": "AAPapAv4LbFbiVawEjagUBluYqN5rhna-8nuldDvOx8"
}
```

Figure 6: AES 256-bit symmetric encryption key

The field "kty" value of "oct" identifies this as a symmetric key. The field "k" value is the symmetric key.

For the content encryption algorithm "A256GCM", the field "k" value is exactly 32 octets in length when decoded, padded with leading zero (0x00) octets to reach the expected length.

4. JSON Web Signature Examples

The following sections demonstrate how to generate various JWS objects.

All of the succeeding examples use the following payload plaintext (an abridged quote from "The Fellowship of the Ring" [LOTR-FELLOWSHIP]), serialized as UTF-8. The sequence "\xe2\x80\x99" is substituted for (U+2019 RIGHT SINGLE QUOTATION MARK), and line breaks (U+000A LINE FEED) replace some " " (U+0020 SPACE) to improve readability:

```
It\xe2\x80\x99s a dangerous business, Frodo, going out your
door. You step onto the road, and if you don’t keep your feet,
there\xe2\x80\x99s no knowing where you might be swept off
to.
```

Figure 7: Payload content plaintext

The Payload - with the sequence "\xe2\x80\x99" replaced with (U+2019 RIGHT SINGLE QUOTATION MARK) and line breaks (U+000A LINE FEED) replaced with " " (U+0020 SPACE) - encoded as UTF-8 then as [RFC4648] base64url:

```
SXTigJlzIGEgZGFuZ2Vyb3VzIGJ1c2luZXNzLCBGcm9kbywgZ29pbmcgb3V0IH
lvdXIgZG9vci4gWW91IHN0ZXAgb250byB0aGUgcm9hZCwgYW5kIGlmIHlvdSBk
b24ndCBrZWVwIHlvdXIgZmVldCwgdGhlcmXigJlzIG5vIGtub3dpbmcgd2hlcm
UgeW91IG1pZ2h0IGJlIHN3ZXB0IG9mZiB0by4
```

Figure 8: Payload content, base64url-encoded

4.1. RSA v1.5 Signature

This example illustrates signing content using the "RS256" (RSASSA-PKCS1-v1_5 with SHA-256) algorithm.

Note that whitespace is added for readability as described in Section 1.1.

4.1.1. Input Factors

The following are supplied before beginning the signing operation:

- Payload content; this example uses the content from Figure 7, encoded using [RFC4648] base64url to produce Figure 8.
- RSA private key; this example uses the key from Figure 4.
- "alg" parameter of "RS256".

4.1.2. Signing Operation

The following are generated to complete the signing operation:

- JWS Protected Header; this example uses the header from Figure 9, encoded using [RFC4648] base64url to produce Figure 10.

```
{
  "alg": "RS256",
  "kid": "bilbo.baggins@hobbiton.example"
}
```

Figure 9: JWS Protected Header JSON

```
eyJhbGciOiJSUzI1NiIsImtpZCI6ImJpbGJvLmJhZ2dpbnNAaG9iYml0b24uZX
hhbXBsZSJ9
```

Figure 10: JWS Protected Header, base64url-encoded

The JWS Protected Header (Figure 10) and Payload content (Figure 8) are combined as described in section 5.1 of [I-D.ietf-jose-json-web-signature] to produce the JWS Signing Input Figure 11.

```
eyJhbGciOiJSUzI1NiIsImtpZCI6ImJpbGJvLmJhZ2dpbnNAaG9iYml0b24uZX
hhbXBsZSJ9
.
SXTigJlzIGEgZGFuZ2Vyb3VzIGJ1c2luZXNzLCBGcm9kbywgZ29pbmcgb3V0IH
lvdXIgZG9vci4gWW91IHN0ZXAgb250byB0aGUgcm9hZCwgYW5kIGlmIHlvdSBk
b24ndCBrZWVwIHlvdXIgZmVldCwgdGhlcmXigJlzIG5vIGtub3dpbmcgd2hlcm
UgeW91IG1pZ2h0IGJlIHN3ZXB0IG9mZiB0by4
```

Figure 11: JWS Signing Input

Performing the signature operation over the JWS Signing Input (Figure 11) produces the JWS Signature (Figure 12).

```
MRjdkly7_-oTPTS3AXP41iQIGKa80A0ZmTuV5MEaHoxnW2e5CZ5NlKtainoFmK
ZopdHM1O2U4mwzJdQx996ivp83xuglII7PNDi84wnB-BDkoBwA78185hX-Es4J
IwmDLJK3lfWRa-XtL0RnltuYv746iYTh_qHRD68BNt1uSNCrUCTJDt5aAE6x8w
W1Kt9eRo4QPocSadnHXFxnt8Is9UzpERV0ePPQdLuW3IS_de3xyIrDaLGdjluP
xUAhb6L2aXic1U12podGU0KLUQSE_oI-ZnmKJ3F4uOZDnd6QZWJushZ41Axf_f
cIe8u9ipH84ogoree7vjbU5y18kDquDg
```

Figure 12: JWS Signature, base64url-encoded

4.1.3. Output Results

The following compose the resulting JWS object:

- JWS Protected Header (Figure 9)
- Payload content (Figure 8)
- Signature (Figure 12)

The resulting JWS object using the Compact serialization:

```
eyJhbGciOiJSUzI1NiIsImtpZCI6ImJpbGJvLmJhZ2dpbnNAaG9iYml0b24uZX
hhbXBsZSJ9
.
SXTigJlzIGEgZGFuZ2Vyb3VzIGJ1c2luZXNzLCBGcm9kbywgZ29pbmcgb3V0IH
lvdXIgZG9vci4gWW91IHN0ZXAgb250byB0aGUgcm9hZCwgYW5kIGlmIHlvdSBk
b24ndCBrZWVwIHlvdXIgZmVldCwgdGhlcmXigJlzIG5vIGtub3dpbmcgd2hlcm
UgeW91IG1pZ2h0IGJlIHN3ZXB0IG9mZiB0by4
.
MRjdkly7_-oTPTS3AXP41iQIGKa80A0ZmTuV5MEaHoxnW2e5CZ5NlKtainoFmK
ZopdHM1O2U4mwzJdQx996ivp83xuglII7PNDi84wnB-BDkoBwA78185hX-Es4J
IwmDLJK3lfWRa-XtL0RnltuYv746iYTh_qHRD68BNt1uSNCrUCTJDt5aAE6x8w
W1Kt9eRo4QPocSadnHXFxnt8Is9UzpERV0ePPQdLuW3IS_de3xyIrDaLGdjluP
xUAhb6L2aXic1U12podGU0KLUQSE_oI-ZnmKJ3F4uOZDnd6QZWJushZ41Axf_f
cIe8u9ipH84ogoree7vjbU5y18kDquDg
```

Figure 13: Compact Serialization

The resulting JWS object using the JSON General Serialization:

```
{
  "payload": "SXTigJlzIGEgZGFuZ2Vyb3VzIGJ1c2luZXNzLCBGcm9kbywg
      Z29pbmcgb3V0IHlvdXIgZG9vci4gWW91IHN0ZXAgb250byB0aGUgcm9h
      ZCwgYW5kIGlmIHlvdSBkb24ndCBrZWVwIHlvdXIgZmVldCwgdGhlcmXi
      gJlzIG5vIGtub3dpbmcgd2hlcmUgeW91IG1pZ2h0IGJlIHN3ZXB0IG9m
      ZiB0by4",
  "signatures": [
    {
      "protected": "eyJhbGciOiJSUzI1NiIsImtpZCI6ImJpbGJvLmJhZ2
          dpbnNAaG9iYml0b24uZXhhbXBsZSJ9",
      "signature": "MRjdkly7_-oTPTS3AXP41iQIGKa80A0ZmTuV5MEaHo
          xnW2e5CZ5NlKtainoFmKZopdHM1O2U4mwzJdQx996ivp83xuglII
          7PNDi84wnB-BDkoBwA78185hX-Es4JIwmDLJK3lfWRa-XtL0Rnlt
          uYv746iYTh_qHRD68BNt1uSNCrUCTJDt5aAE6x8wW1Kt9eRo4QPo
          cSadnHXFxnt8Is9UzpERV0ePPQdLuW3IS_de3xyIrDaLGdjluPxU
          Ahb6L2aXic1U12podGU0KLUQSE_oI-ZnmKJ3F4uOZDnd6QZWJush
          Z41Axf_fcIe8u9ipH84ogoree7vjbU5y18kDquDg"
    }
  ]
}
```

Figure 14: JSON General Serialization

The resulting JWS object using the JSON Flattened Serialization:

```
{
  "payload": "SXTigJlzIGEgZGFuZ2Vyb3VzIGJ1c2luZXNzLCBGcm9kbywg
      Z29pbmcgb3V0IHlvdXIgZG9vci4gWW91IHN0ZXAgb250byB0aGUgcm9h
      ZCwgYW5kIGlmIHlvdSBkb24ndCBrZWVwIHlvdXIgZmVldCwgdGhlcmXi
      gJlzIG5vIGtub3dpbmcgd2hlcmUgeW91IG1pZ2h0IGJlIHN3ZXB0IG9m
      ZiB0by4",
  "protected": "eyJhbGciOiJSUzI1NiIsImtpZCI6ImJpbGJvLmJhZ2dpbn
      NAaG9iYml0b24uZXhhbXBsZSJ9",
  "signature": "MRjdkly7_-oTPTS3AXP41iQIGKa80A0ZmTuV5MEaHoxnW2
      e5CZ5NlKtainoFmKZopdHM1O2U4mwzJdQx996ivp83xuglII7PNDi84w
      nB-BDkoBwA78185hX-Es4JIwmDLJK3lfWRa-XtL0RnltuYv746iYTh_q
      HRD68BNt1uSNCrUCTJDt5aAE6x8wW1Kt9eRo4QPocSadnHXFxnt8Is9U
      zpERV0ePPQdLuW3IS_de3xyIrDaLGdjluPxUAhb6L2aXic1U12podGU0
      KLUQSE_oI-ZnmKJ3F4uOZDnd6QZWJushZ41Axf_fcIe8u9ipH84ogore
      e7vjbU5y18kDquDg"
}
```

Figure 15: JSON Flattened Serialization

4.2. RSA-PSS Signature

This example illustrates signing content using the "PS384" (RSASSA-PSS with SHA-384) algorithm.

Note that RSASSA-PSS uses random data to generate the signature; it might not be possible to exactly replicate the results in this section.

Note that whitespace is added for readability as described in Section 1.1.

4.2.1. Input Factors

The following are supplied before beginning the signing operation:

- Payload content; this example uses the content from Figure 7, encoded using [RFC4648] base64url to produce Figure 8.
- RSA private key; this example uses the key from Figure 4.
- "alg" parameter of "PS384".

4.2.2. Signing Operation

The following are generated to complete the signing operation:

- JWS Protected Header; this example uses the header from Figure 16, encoded using [RFC4648] base64url to produce Figure 17.

```
{
  "alg": "PS384",
  "kid": "bilbo.baggins@hobbiton.example"
}
```

Figure 16: JWS Protected Header JSON

```
eyJhbGciOiJQUzM4NCIsImtpZCI6ImJpbGJvLmJhZ2dpbnNAaG9iYml0b24uZX
hhbXBsZSJ9
```

Figure 17: JWS Protected Header, base64url-encoded

The JWS Protected Header (Figure 17) and Payload content (Figure 8) are combined as described in [I-D.ietf-jose-json-web-signature] to produce the JWS Signing Input Figure 18.

```
eyJhbGciOiJQUzM4NCIsImtpZCI6ImJpbGJvLmJhZ2dpbnNAaG9iYml0b24uZX
hhbXBsZSJ9
.
SXTigJlzIGEgZGFuZ2Vyb3VzIGJ1c2luZXNzLCBGcm9kbywgZ29pbmcgb3V0IH
lvdXIgZG9vci4gWW91IHN0ZXAgb250byB0aGUgcm9hZCwgYW5kIGlmIHlvdSBk
b24ndCBrZWVwIHlvdXIgZmVldCwgdGhlcmXigJlzIG5vIGtub3dpbmcgd2hlcm
UgeW91IG1pZ2h0IGJlIHN3ZXB0IG9mZiB0by4
```

Figure 18: JWS Signing Input

Performing the signature operation over the JWS Signing Input (Figure 18) produces the JWS Signature (Figure 19).

```
cu22eBqkYDKgIlTpzDXGvaFfz6WGoz7fUDcfT0kkOy42miAh2qyBzk1xEsnk2I
pN6-tPid6VrklHkqsGqDqHCdP6O8TTB5dDDItllVo6_1OLPpcbUrhiUSMxbbXU
vdvWXzg-UD8biiReQFlfz28zGWVsdiNAUf8ZnyPEgVFn442ZdNqiVJRmBqrYRX
e8P_ijQ7p8Vdz0TTrxUeT3lm8d9shnr2lfJT8ImUjvAA2Xez2Mlp8cBE5awDzT
0qI0n6uiP1aCN_2_jLAeQTlqRHtfa64QQSUmFAAjVKPbByi7xho0uTOcbH510a
6GYmJUAfmWjwZ6oD4ifKo8DYM-X72Eaw
```

Figure 19: JWS Signature, base64url-encoded

4.2.3. Output Results

The following compose the resulting JWS object:

- JWS Protected Header (Figure 17)
- Payload content (Figure 8)
- Signature (Figure 19)

The resulting JWS object using the Compact serialization:

```
eyJhbGciOiJQUzM4NCIsImtpZCI6ImJpbGJvLmJhZ2dpbnNAaG9iYml0b24uZX
hhbXBsZSJ9
.
SXTigJlzIGEgZGFuZ2Vyb3VzIGJ1c2luZXNzLCBGcm9kbywgZ29pbmcgb3V0IH
lvdXIgZG9vci4gWW91IHN0ZXAgb250byB0aGUgcm9hZCwgYW5kIGlmIHlvdSBk
b24ndCBrZWVwIHlvdXIgZmVldCwgdGhlcmXigJlzIG5vIGtub3dpbmcgd2hlcm
UgeW91IG1pZ2h0IGJlIHN3ZXB0IG9mZiB0by4
.
cu22eBqkYDKgIlTpzDXGvaFfz6WGoz7fUDcfT0kkOy42miAh2qyBzk1xEsnk2I
pN6-tPid6VrklHkqsGqDqHCdP6O8TTB5dDDItllVo6_1OLPpcbUrhiUSMxbbXU
vdvWXzg-UD8biiReQFlfz28zGWVsdiNAUf8ZnyPEgVFn442ZdNqiVJRmBqrYRX
e8P_ijQ7p8Vdz0TTrxUeT3lm8d9shnr2lfJT8ImUjvAA2Xez2Mlp8cBE5awDzT
0qI0n6uiP1aCN_2_jLAeQTlqRHtfa64QQSUmFAAjVKPbByi7xho0uTOcbH510a
6GYmJUAfmWjwZ6oD4ifKo8DYM-X72Eaw
```

Figure 20: Compact Serialization

The resulting JWS object using the JSON General Serialization:

```
{
  "payload": "SXTigJlzIGEgZGFuZ2Vyb3VzIGJ1c2luZXNzLCBGcm9kbywg
      Z29pbmcgb3V0IHlvdXIgZG9vci4gWW91IHN0ZXAgb250byB0aGUgcm9h
      ZCwgYW5kIGlmIHlvdSBkb24ndCBrZWVwIHlvdXIgZmVldCwgdGhlcmXi
      gJlzIG5vIGtub3dpbmcgd2hlcmUgeW91IG1pZ2h0IGJlIHN3ZXB0IG9m
      ZiB0by4",
  "signatures": [
    {
      "protected": "eyJhbGciOiJQUzM4NCIsImtpZCI6ImJpbGJvLmJhZ2
          dpbnNAaG9iYml0b24uZXhhbXBsZSJ9",
      "signature": "cu22eBqkYDKgIlTpzDXGvaFfz6WGoz7fUDcfT0kkOy
          42miAh2qyBzk1xEsnk2IpN6-tPid6VrklHkqsGqDqHCdP6O8TTB5
          dDDItllVo6_1OLPpcbUrhiUSMxbbXUvdvWXzg-UD8biiReQFlfz2
          8zGWVsdiNAUf8ZnyPEgVFn442ZdNqiVJRmBqrYRXe8P_ijQ7p8Vd
          z0TTrxUeT3lm8d9shnr2lfJT8ImUjvAA2Xez2Mlp8cBE5awDzT0q
          I0n6uiP1aCN_2_jLAeQTlqRHtfa64QQSUmFAAjVKPbByi7xho0uT
          OcbH510a6GYmJUAfmWjwZ6oD4ifKo8DYM-X72Eaw"
    }
  ]
}
```

Figure 21: JSON General Serialization

The resulting JWS object using the JSON Flattened Serialization:

```
{
  "payload": "SXTigJlzIGEgZGFuZ2Vyb3VzIGJ1c2luZXNzLCBGcm9kbywg
      Z29pbmcgb3V0IHlvdXIgZG9vci4gWW91IHN0ZXAgb250byB0aGUgcm9h
      ZCwgYW5kIGlmIHlvdSBkb24ndCBrZWVwIHlvdXIgZmVldCwgdGhlcmXi
      gJlzIG5vIGtub3dpbmcgd2hlcmUgeW91IG1pZ2h0IGJlIHN3ZXB0IG9m
      ZiB0by4",
  "protected": "eyJhbGciOiJQUzM4NCIsImtpZCI6ImJpbGJvLmJhZ2dpbn
      NAaG9iYml0b24uZXhhbXBsZSJ9",
  "signature": "cu22eBqkYDKgIlTpzDXGvaFfz6WGoz7fUDcfT0kkOy42mi
      Ah2qyBzk1xEsnk2IpN6-tPid6VrklHkqsGqDqHCdP6O8TTB5dDDItllV
      o6_1OLPpcbUrhiUSMxbbXUvdvWXzg-UD8biiReQFlfz28zGWVsdiNAUf
      8ZnyPEgVFn442ZdNqiVJRmBqrYRXe8P_ijQ7p8Vdz0TTrxUeT3lm8d9s
      hnr2lfJT8ImUjvAA2Xez2Mlp8cBE5awDzT0qI0n6uiP1aCN_2_jLAeQT
      lqRHtfa64QQSUmFAAjVKPbByi7xho0uTOcbH510a6GYmJUAfmWjwZ6oD
      4ifKo8DYM-X72Eaw"
}
```

Figure 22: JSON Flattened Serialization

4.3. ECDSA Signature

This example illustrates signing content using the "ES512" (ECDSA with curve P-521 and SHA-512) algorithm.

Note that ECDSA uses random data to generate the signature; it might not be possible to exactly replicate the results in this section.

Note that whitespace is added for readability as described in Section 1.1.

4.3.1. Input Factors

The following are supplied before beginning the signing operation:

- Payload content; this example uses the content from Figure 7, encoded using [RFC4648] base64url to produce Figure 8.
- EC private key on the curve P-521; this example uses the key from Figure 2.
- "alg" parameter of "ES512"

4.3.2. Signing Operation

The following are generated before beginning the signature process:

- JWS Protected Header; this example uses the header from Figure 23, encoded using [RFC4648] base64url to produce Figure 24.

```
{
  "alg": "ES512",
  "kid": "bilbo.baggins@hobbiton.example"
}
```

Figure 23: JWS Protected Header JSON

```
eyJhbGciOiJFUzUxMiIsImtpZCI6ImJpbGJvLmJhZ2dpbnNAaG9iYml0b24uZX
hhbXBsZSJ9
```

Figure 24: JWS Protected Header, base64url-encoded

The JWS Protected Header (Figure 24) and Payload content (Figure 8) are combined as described in [I-D.ietf-jose-json-web-signature] to produce the JWS Signing Input Figure 25.

```
eyJhbGciOiJFUzUxMiIsImtpZCI6ImJpbGJvLmJhZ2dpbnNAaG9iYml0b24uZX
hhbXBsZSJ9
.
SXTigJlzIGEgZGFuZ2Vyb3VzIGJ1c2luZXNzLCBGcm9kbywgZ29pbmcgb3V0IH
lvdXIgZG9vci4gWW91IHN0ZXAgb250byB0aGUgcm9hZCwgYW5kIGlmIHlvdSBk
b24ndCBrZWVwIHlvdXIgZmVldCwgdGhlcmXigJlzIG5vIGtub3dpbmcgd2hlcm
UgeW91IG1pZ2h0IGJlIHN3ZXB0IG9mZiB0by4
```

Figure 25: JWS Signing Input

Performing the signature operation over the JWS Signing Input (Figure 25) produces the JWS Signature (Figure 26).

```
AE_R_YZCChjn4791jSQCrdPZCNYqHXCTZH0-JZGYNlaAjP2kqaluUIIUnC9qvb
u9Plon7KRTzoNEuT4Va2cmL1eJAQy3mtPBu_u_sDDyYjnAMDxXPn7XrT0lw-kv
AD890jl8e2puQens_IEKBpHABlsbEPX6sFY8OcGDqoRuBomu9xQ2
```

Figure 26: JWS Signature, base64url-encoded

4.3.3. Output Results

The following compose the resulting JWS object:

- JWS Protected Header (Figure 24)
- Payload content (Figure 8)
- Signature (Figure 26)

The resulting JWS object using the Compact serialization:

```
eyJhbGciOiJFUzUxMiIsImtpZCI6ImJpbGJvLmJhZ2dpbnNAaG9iYml0b24uZX
hhbXBsZSJ9
.
SXTigJlzIGEgZGFuZ2Vyb3VzIGJ1c2luZXNzLCBGcm9kbywgZ29pbmcgb3V0IH
lvdXIgZG9vci4gWW91IHN0ZXAgb250byB0aGUgcm9hZCwgYW5kIGlmIHlvdSBk
b24ndCBrZWVwIHlvdXIgZmVldCwgdGhlcmXigJlzIG5vIGtub3dpbmcgd2hlcm
UgeW91IG1pZ2h0IGJlIHN3ZXB0IG9mZiB0by4
.
AE_R_YZCChjn4791jSQCrdPZCNYqHXCTZH0-JZGYNlaAjP2kqaluUIIUnC9qvb
u9Plon7KRTzoNEuT4Va2cmL1eJAQy3mtPBu_u_sDDyYjnAMDxXPn7XrT0lw-kv
AD890jl8e2puQens_IEKBpHABlsbEPX6sFY8OcGDqoRuBomu9xQ2
```

Figure 27: Compact Serialization

The resulting JWS object using the JSON General Serialization:

```
{
  "payload": "SXTigJlzIGEgZGFuZ2Vyb3VzIGJ1c2luZXNzLCBGcm9kbywg
      Z29pbmcgb3V0IHlvdXIgZG9vci4gWW91IHN0ZXAgb250byB0aGUgcm9h
      ZCwgYW5kIGlmIHlvdSBkb24ndCBrZWVwIHlvdXIgZmVldCwgdGhlcmXi
      gJlzIG5vIGtub3dpbmcgd2hlcmUgeW91IG1pZ2h0IGJlIHN3ZXB0IG9m
      ZiB0by4",
  "signatures": [
    {
      "protected": "eyJhbGciOiJFUzUxMiIsImtpZCI6ImJpbGJvLmJhZ2
          dpbnNAaG9iYml0b24uZXhhbXBsZSJ9",
      "signature": "AE_R_YZCChjn4791jSQCrdPZCNYqHXCTZH0-JZGYNl
          aAjP2kqaluUIIUnC9qvbu9Plon7KRTzoNEuT4Va2cmL1eJAQy3mt
          PBu_u_sDDyYjnAMDxXPn7XrT0lw-kvAD890jl8e2puQens_IEKBp
          HABlsbEPX6sFY8OcGDqoRuBomu9xQ2"
    }
  ]
}
```

Figure 28: JSON General Serialization

The resulting JWS object using the JSON Flattened Serialization:

```
{
  "payload": "SXTigJlzIGEgZGFuZ2Vyb3VzIGJ1c2luZXNzLCBGcm9kbywg
      Z29pbmcgb3V0IHlvdXIgZG9vci4gWW91IHN0ZXAgb250byB0aGUgcm9h
      ZCwgYW5kIGlmIHlvdSBkb24ndCBrZWVwIHlvdXIgZmVldCwgdGhlcmXi
      gJlzIG5vIGtub3dpbmcgd2hlcmUgeW91IG1pZ2h0IGJlIHN3ZXB0IG9m
      ZiB0by4",
  "protected": "eyJhbGciOiJFUzUxMiIsImtpZCI6ImJpbGJvLmJhZ2dpbn
      NAaG9iYml0b24uZXhhbXBsZSJ9",
  "signature": "AE_R_YZCChjn4791jSQCrdPZCNYqHXCTZH0-JZGYNlaAjP
      2kqaluUIIUnC9qvbu9Plon7KRTzoNEuT4Va2cmL1eJAQy3mtPBu_u_sD
      DyYjnAMDxXPn7XrT0lw-kvAD890jl8e2puQens_IEKBpHABlsbEPX6sF
      Y8OcGDqoRuBomu9xQ2"
}
```


Figure 29: JSON Flattened Serialization

4.4. HMAC-SHA2 Integrity Protection

This example illustrates integrity protecting content using the "HS256" (HMAC-SHA-256) algorithm.

Note that whitespace is added for readability as described in Section 1.1.

4.4.1. Input Factors

The following are supplied before beginning the signing operation:

- Payload content; this example uses the content from Figure 7, encoded using [RFC4648] base64url to produce Figure 8.
- HMAC symmetric key; this example uses the key from Figure 5.
- "alg" parameter of "HS256".

4.4.2. Signing Operation

The following are generated before completing the signing operation:

- JWS Protected Header; this example uses the header from Figure 30, encoded using [RFC4648] base64url to produce Figure 31.


```
{
  "alg": "HS256",
  "kid": "018c0ae5-4d9b-471b-bfd6-eef314bc7037"
}
```


Figure 30: JWS Protected Header JSON

```
eyJhbGciOiJIUzI1NiIsImtpZCI6IjAxOGMwYWU1LTRkOWItNDcxYi1iZmQ2LW
VlZjMxNGJjNzAzNyJ9
```

Figure 31: JWS Protected Header, base64url-encoded

The JWS Protected Header (Figure 31) and Payload content (Figure 8) are combined as described in [I-D.ietf-jose-json-web-signature] to produce the JWS Signing Input Figure 32.

```
eyJhbGciOiJIUzI1NiIsImtpZCI6IjAxOGMwYWU1LTRkOWItNDcxYi1iZmQ2LW
VlZjMxNGJjNzAzNyJ9
.
SXTigJlzIGEgZGFuZ2Vyb3VzIGJ1c2luZXNzLCBGcm9kbywgZ29pbmcgb3V0IH
lvdXIgZG9vci4gWW91IHN0ZXAgb250byB0aGUgcm9hZCwgYW5kIGlmIHlvdSBk
b24ndCBrZWVwIHlvdXIgZmVldCwgdGhlcmXigJlzIG5vIGtub3dpbmcgd2hlcm
UgeW91IG1pZ2h0IGJlIHN3ZXB0IG9mZiB0by4
```

Figure 32: JWS Signing Input

Performing the signature operation over the JWS Signing Input (Figure 32) produces the JWS Signature (Figure 33).

```
s0h6KThzkfBBBkLspW1h84VsJZFTsPPqMDA7g1Md7p0
```

Figure 33: JWS Signature, base64url-encoded

4.4.3. Output Results

The following compose the resulting JWS object:

- JWS Protected Header (Figure 31)
- Payload content (Figure 8)
- Signature (Figure 33)

The resulting JWS object using the Compact serialization:

```
eyJhbGciOiJIUzI1NiIsImtpZCI6IjAxOGMwYWU1LTRkOWItNDcxYi1iZmQ2LW
VlZjMxNGJjNzAzNyJ9
.
SXTigJlzIGEgZGFuZ2Vyb3VzIGJ1c2luZXNzLCBGcm9kbywgZ29pbmcgb3V0IH
lvdXIgZG9vci4gWW91IHN0ZXAgb250byB0aGUgcm9hZCwgYW5kIGlmIHlvdSBk
b24ndCBrZWVwIHlvdXIgZmVldCwgdGhlcmXigJlzIG5vIGtub3dpbmcgd2hlcm
UgeW91IG1pZ2h0IGJlIHN3ZXB0IG9mZiB0by4
.
s0h6KThzkfBBBkLspW1h84VsJZFTsPPqMDA7g1Md7p0
```

Figure 34: Compact Serialization

The resulting JWS object using the JSON General Serialization:

```
{
  "payload": "SXTigJlzIGEgZGFuZ2Vyb3VzIGJ1c2luZXNzLCBGcm9kbywg
      Z29pbmcgb3V0IHlvdXIgZG9vci4gWW91IHN0ZXAgb250byB0aGUgcm9h
      ZCwgYW5kIGlmIHlvdSBkb24ndCBrZWVwIHlvdXIgZmVldCwgdGhlcmXi
      gJlzIG5vIGtub3dpbmcgd2hlcmUgeW91IG1pZ2h0IGJlIHN3ZXB0IG9m
      ZiB0by4",
  "signatures": [
    {
      "protected": "eyJhbGciOiJIUzI1NiIsImtpZCI6IjAxOGMwYWU1LT
          RkOWItNDcxYi1iZmQ2LWVlZjMxNGJjNzAzNyJ9",
      "signature": "s0h6KThzkfBBBkLspW1h84VsJZFTsPPqMDA7g1Md7p
          0"
    }
  ]
}
```

Figure 35: JSON General Serialization

The resulting JWS object using the JSON Flattened Serialization:

```
{
  "payload": "SXTigJlzIGEgZGFuZ2Vyb3VzIGJ1c2luZXNzLCBGcm9kbywg
      Z29pbmcgb3V0IHlvdXIgZG9vci4gWW91IHN0ZXAgb250byB0aGUgcm9h
      ZCwgYW5kIGlmIHlvdSBkb24ndCBrZWVwIHlvdXIgZmVldCwgdGhlcmXi
      gJlzIG5vIGtub3dpbmcgd2hlcmUgeW91IG1pZ2h0IGJlIHN3ZXB0IG9m
      ZiB0by4",
  "protected": "eyJhbGciOiJIUzI1NiIsImtpZCI6IjAxOGMwYWU1LTRkOW
      ItNDcxYi1iZmQ2LWVlZjMxNGJjNzAzNyJ9",
  "signature": "s0h6KThzkfBBBkLspW1h84VsJZFTsPPqMDA7g1Md7p0"
}
```

Figure 36: JSON Flattened Serialization

4.5. Signature with Detached Content

This example illustrates a signature with detached content. This example is identical others, except the resulting JWS objects do not include the Payload field. Instead, the application is expected to locate it elsewhere. For example, the signature might be in a meta-data section, with the payload being the content.

Note that whitespace is added for readability as described in Section 1.1.

4.5.1. Input Factors

The following are supplied before beginning the signing operation:

- Payload content; this example uses the content from Figure 7, encoded using [RFC4648] base64url to produce Figure 8.
- Signing key; this example uses the AES symmetric key from Figure 5.
- Signing algorithm; this example uses "HS256".

4.5.2. Signing Operation

The following are generated before completing the signing operation:

- JWS Protected Header; this example uses the header from Figure 37, encoded using [RFC4648] base64url to produce Figure 38.

The JWS Protected Header parameters:

```
{
  "alg": "HS256",
  "kid": "018c0ae5-4d9b-471b-bfd6-eef314bc7037"
}
```

Figure 37: JWS Protected Header JSON

```
eyJhbGciOiJIUzI1NiIsImtpZCI6IjAxOGMwYWU1LTRkOWItNDcxYi1iZmQ2LW
VlZjMxNGJjNzAzNyJ9
```

Figure 38: JWS Protected Header, base64url-encoded

The JWS Protected Header (Figure 38) and Payload content (Figure 8) are combined as described in [I-D.ietf-jose-json-web-signature] to produce the JWS Signing Input Figure 39.

```
eyJhbGciOiJIUzI1NiIsImtpZCI6IjAxOGMwYWU1LTRkOWItNDcxYi1iZmQ2LW
VlZjMxNGJjNzAzNyJ9
.
SXTigJlzIGEgZGFuZ2Vyb3VzIGJ1c2luZXNzLCBGcm9kbywgZ29pbmcgb3V0IH
lvdXIgZG9vci4gWW91IHN0ZXAgb250byB0aGUgcm9hZCwgYW5kIGlmIHlvdSBk
b24ndCBrZWVwIHlvdXIgZmVldCwgdGhlcmXigJlzIG5vIGtub3dpbmcgd2hlcm
UgeW91IG1pZ2h0IGJlIHN3ZXB0IG9mZiB0by4
```

Figure 39: JWS Signing Input

Performing the signature operation over the JWS Signing Input (Figure 39) produces the JWS Signature (Figure 40).

```
s0h6KThzkfBBBkLspW1h84VsJZFTsPPqMDA7g1Md7p0
```

Figure 40: JWS Signature, base64url-encoded

4.5.3. Output Results

The following compose the resulting JWS object:

- JWS Protected Header (Figure 38)
- Signature (Figure 40)

The resulting JWS object using the Compact serialization:

```
eyJhbGciOiJIUzI1NiIsImtpZCI6IjAxOGMwYWU1LTRkOWItNDcxYi1iZmQ2LW
VlZjMxNGJjNzAzNyJ9
.
.
s0h6KThzkfBBBkLspW1h84VsJZFTsPPqMDA7g1Md7p0
```

Figure 41: JSON General Serialization

The resulting JWS object using the JSON General Serialization:

```
{
  "signatures": [
    {
      "protected": "eyJhbGciOiJIUzI1NiIsImtpZCI6IjAxOGMwYWU1LT
          RkOWItNDcxYi1iZmQ2LWVlZjMxNGJjNzAzNyJ9",
      "signature": "s0h6KThzkfBBBkLspW1h84VsJZFTsPPqMDA7g1Md7p
          0"
    }
  ]
}
```

Figure 42: JSON General Serialization

The resulting JWS object using the JSON Flattened Serialization:

```
{
  "protected": "eyJhbGciOiJIUzI1NiIsImtpZCI6IjAxOGMwYWU1LTRkOW
      ItNDcxYi1iZmQ2LWVlZjMxNGJjNzAzNyJ9",
  "signature": "s0h6KThzkfBBBkLspW1h84VsJZFTsPPqMDA7g1Md7p0"
}
```

Figure 43: JSON Flattened Serialization

4.6. Protecting Specific Header Fields

This example illustrates a signature where only certain header parameters are protected. Since this example contains both unprotected and protected header parameters, only the JSON General Serialization and JSON Flattened Serialization are possible.

Note that whitespace is added for readability as described in Section 1.1.

4.6.1. Input Factors

The following are supplied before beginning the signing operation:

- Payload content; this example uses the content from Figure 7, encoded using [RFC4648] base64url to produce Figure 8.
- Signing key; this example uses the AES symmetric key from Figure 5.
- Signing algorithm; this example uses "HS256".

4.6.2. Signing Operation

The following are generated before completing the signing operation:

- JWS Protected Header; this example uses the header from Figure 44, encoded using [RFC4648] base64url to produce Figure 45.
- JWS unprotected Header; this example uses the header from Figure 46.

The JWS Protected Header parameters:

```
{
  "alg": "HS256"
}
```

Figure 44: JWS Protected Header JSON

```
eyJhbGciOiJIUzI1NiJ9
```

Figure 45: JWS Protected Header, base64url-encoded

```
{
  "kid": "018c0ae5-4d9b-471b-bfd6-eef314bc7037"
}
```

Figure 46: JWS Unprotected Header JSON

The JWS Protected Header (Figure 45) and Payload content (Figure 8) are combined as described in [I-D.ietf-jose-json-web-signature] to produce the JWS Signing Input Figure 47.

```
eyJhbGciOiJIUzI1NiJ9
.
SXTigJlzIGEgZGFuZ2Vyb3VzIGJ1c2luZXNzLCBGcm9kbywgZ29pbmcgb3V0IH
lvdXIgZG9vci4gWW91IHN0ZXAgb250byB0aGUgcm9hZCwgYW5kIGlmIHlvdSBk
b24ndCBrZWVwIHlvdXIgZmVldCwgdGhlcmXigJlzIG5vIGtub3dpbmcgd2hlcm
UgeW91IG1pZ2h0IGJlIHN3ZXB0IG9mZiB0by4
```

Figure 47: JWS Signing Input

Performing the signature operation over the JWS Signing Input (Figure 47) produces the JWS Signature (Figure 48).

```
bWUSVaxorn7bEF1djytBd0kHv70Ly5pvbomzMWSOr20
```

Figure 48: JWS Signature, base64url-encoded

4.6.3. Output Results

The following compose the resulting JWS object:

- JWS Protected Header (Figure 45)
- JWS Unprotected Header (Figure 46)
- Payload content (Figure 8)
- Signature (Figure 48)

The Compact Serialization is not presented because it does not support this use case.

The resulting JWS object using the JSON General Serialization:

```
{
  "payload": "SXTigJlzIGEgZGFuZ2Vyb3VzIGJ1c2luZXNzLCBGcm9kbywg
      Z29pbmcgb3V0IHlvdXIgZG9vci4gWW91IHN0ZXAgb250byB0aGUgcm9h
      ZCwgYW5kIGlmIHlvdSBkb24ndCBrZWVwIHlvdXIgZmVldCwgdGhlcmXi
      gJlzIG5vIGtub3dpbmcgd2hlcmUgeW91IG1pZ2h0IGJlIHN3ZXB0IG9m
      ZiB0by4",
  "signatures": [
    {
      "protected": "eyJhbGciOiJIUzI1NiJ9",
      "header": {
        "kid": "018c0ae5-4d9b-471b-bfd6-eef314bc7037"
      },
      "signature": "bWUSVaxorn7bEF1djytBd0kHv70Ly5pvbomzMWSOr2
          0"
    }
  ]
}
```

Figure 49: JSON General Serialization

The resulting JWS object using the JSON Flattened Serialization:

```
{
  "payload": "SXTigJlzIGEgZGFuZ2Vyb3VzIGJ1c2luZXNzLCBGcm9kbywg
      Z29pbmcgb3V0IHlvdXIgZG9vci4gWW91IHN0ZXAgb250byB0aGUgcm9h
      ZCwgYW5kIGlmIHlvdSBkb24ndCBrZWVwIHlvdXIgZmVldCwgdGhlcmXi
      gJlzIG5vIGtub3dpbmcgd2hlcmUgeW91IG1pZ2h0IGJlIHN3ZXB0IG9m
      ZiB0by4",
  "protected": "eyJhbGciOiJIUzI1NiJ9",
  "header": {
    "kid": "018c0ae5-4d9b-471b-bfd6-eef314bc7037"
  },
  "signature": "bWUSVaxorn7bEF1djytBd0kHv70Ly5pvbomzMWSOr20"
}
```

Figure 50: JSON Flattened Serialization

4.7. Protecting Content Only

This example illustrates a signature where none of the header parameters are protected. Since this example contains only unprotected header parameters, only the JSON General Serialization and JSON Flattened Serialization are possible.

Note that whitespace is added for readability as described in Section 1.1.

4.7.1. Input Factors

The following are supplied before beginning the signing operation:

- Payload content; this example uses the content from Figure 7, encoded using [RFC4648] base64url to produce Figure 8.
- Signing key; this example uses the AES key from Figure 5.
- Signing algorithm; this example uses "HS256"

4.7.2. Signing Operation

The following are generated before completing the signing operation:

- JWS Unprotected Header; this example uses the header from Figure 51.

```
{
  "alg": "HS256",
  "kid": "018c0ae5-4d9b-471b-bfd6-eef314bc7037"
}
```

Figure 51: JWS Unprotected Header JSON

The empty string (as there is no JWS Protected Header) and Payload content (Figure 8) are combined as described in [I-D.ietf-jose-json-web-signature] to produce the JWS Signing Input Figure 52.

```
.
SXTigJlzIGEgZGFuZ2Vyb3VzIGJ1c2luZXNzLCBGcm9kbywgZ29pbmcgb3V0IH
lvdXIgZG9vci4gWW91IHN0ZXAgb250byB0aGUgcm9hZCwgYW5kIGlmIHlvdSBk
b24ndCBrZWVwIHlvdXIgZmVldCwgdGhlcmXigJlzIG5vIGtub3dpbmcgd2hlcm
UgeW91IG1pZ2h0IGJlIHN3ZXB0IG9mZiB0by4
```

Figure 52: JWS Signing Input

Performing the signature operation over the JWS Signing Input (Figure 52) produces the JWS Signature (Figure 53).

```
xuLifqLGiblpv9zBpuZczWhNj1gARaLV3UxvxhJxZuk
```

Figure 53: JWS Signature, base64url-encoded

4.7.3. Output Results

The following compose the resulting JWS object:

- JWS Unprotected Header (Figure 51)
- Payload content (Figure 8)
- Signature (Figure 53)

The Compact Serialization is not presented because it does not support this use case.

The resulting JWS object using the JSON General Serialization:

```
{
  "payload": "SXTigJlzIGEgZGFuZ2Vyb3VzIGJ1c2luZXNzLCBGcm9kbywg
      Z29pbmcgb3V0IHlvdXIgZG9vci4gWW91IHN0ZXAgb250byB0aGUgcm9h
      ZCwgYW5kIGlmIHlvdSBkb24ndCBrZWVwIHlvdXIgZmVldCwgdGhlcmXi
      gJlzIG5vIGtub3dpbmcgd2hlcmUgeW91IG1pZ2h0IGJlIHN3ZXB0IG9m
      ZiB0by4",
  "signatures": [
    {
      "header": {
        "alg": "HS256",
        "kid": "018c0ae5-4d9b-471b-bfd6-eef314bc7037"
      },
      "signature": "xuLifqLGiblpv9zBpuZczWhNj1gARaLV3UxvxhJxZu
          k"
    }
  ]
}
```

Figure 54: JSON General Serialization

The resulting JWS object using the JSON Flattened Serialization:

```
{
  "payload": "SXTigJlzIGEgZGFuZ2Vyb3VzIGJ1c2luZXNzLCBGcm9kbywg
      Z29pbmcgb3V0IHlvdXIgZG9vci4gWW91IHN0ZXAgb250byB0aGUgcm9h
      ZCwgYW5kIGlmIHlvdSBkb24ndCBrZWVwIHlvdXIgZmVldCwgdGhlcmXi
      gJlzIG5vIGtub3dpbmcgd2hlcmUgeW91IG1pZ2h0IGJlIHN3ZXB0IG9m
      ZiB0by4",
  "header": {
    "alg": "HS256",
    "kid": "018c0ae5-4d9b-471b-bfd6-eef314bc7037"
  },
  "signature": "xuLifqLGiblpv9zBpuZczWhNj1gARaLV3UxvxhJxZuk"
}
```

Figure 55: JSON Flattened Serialization

4.8. Multiple Signatures

This example illustrates multiple signatures applied to the same payload. Since this example contains more than one signature, only the JSON serialization is possible.

Note that whitespace is added for readability as described in Section 1.1.

4.8.1. Input Factors

The following are supplied before beginning the signing operation:

- Payload content; this example uses the content from Figure 7, encoded using [RFC4648] base64url to produce Figure 8.
- Signing keys; this example uses the following:
  - RSA private key from Figure 4 for the first signature
  - EC private key from Figure 2 for the second signature
  - AES symmetric key from Figure 5 for the third signature
- Signing algorithms; this example uses the following:
  - "RS256" for the first signature
  - "ES512" for the second signature
  - "HS256" for the third signature

4.8.2. First Signing Operation

The following are generated before completing the first signing operation:

- JWS Protected Header; this example uses the header from Figure 56, encoded using [RFC4648] base64url to produce Figure 57.
- JWS Unprotected Header; this example uses the header from Figure 58.

```
{
  "alg": "RS256"
}
```

Figure 56: Signature #1 JWS Protected Header JSON

```
eyJhbGciOiJSUzI1NiJ9
```

Figure 57: Signature #1 JWS Protected Header, base64url-encoded

```
{
  "kid": "bilbo.baggins@hobbiton.example"
}
```

Figure 58: Signature #1 JWS Unprotected Header JSON

The JWS Protected Header (Figure 57) and Payload content (Figure 8) are combined as described in [I-D.ietf-jose-json-web-signature] to produce the JWS Signing Input Figure 59.

```
eyJhbGciOiJSUzI1NiJ9
.
SXTigJlzIGEgZGFuZ2Vyb3VzIGJ1c2luZXNzLCBGcm9kbywgZ29pbmcgb3V0IH
lvdXIgZG9vci4gWW91IHN0ZXAgb250byB0aGUgcm9hZCwgYW5kIGlmIHlvdSBk
b24ndCBrZWVwIHlvdXIgZmVldCwgdGhlcmXigJlzIG5vIGtub3dpbmcgd2hlcm
UgeW91IG1pZ2h0IGJlIHN3ZXB0IG9mZiB0by4
```

Figure 59: JWS Signing Input

Performing the signature operation over the JWS Signing Input (Figure 59) produces the JWS Signature (Figure 60).

```
MIsjqtVlOpa71KE-Mss8_Nq2YH4FGhiocsqrgi5NvyG53uoimic1tcMdSg-qpt
rzZc7CG6Svw2Y13TDIqHzTUrL_lR2ZFcryNFiHkSw129EghGpwkpxaTn_THJTC
glNbADko1MZBCdwzJxwqZc-1RlpO2HibUYyXSwO97BSe0_evZKdjvvKSgsIqjy
tKSeAMbhMBdMma622_BG5t4sdbuCHtFjp9iJmkio47AIwqkZV1aIZsv33uPUqB
BCXbYoQJwt7mxPftHmNlGoOSMxR_3thmXTCm4US-xiNOyhbm8afKK64jU6_TPt
QHiJeQJxz9G3Tx-083B745_AfYOnlC9w
```

Figure 60: JWS Signature #1, base64url-encoded

The following is the assembled first signature serialized as JSON:

```
{
  "protected": "eyJhbGciOiJSUzI1NiJ9",
  "header": {
    "kid": "bilbo.baggins@hobbiton.example"
  },
  "signature": "MIsjqtVlOpa71KE-Mss8_Nq2YH4FGhiocsqrgi5NvyG53u
      oimic1tcMdSg-qptrzZc7CG6Svw2Y13TDIqHzTUrL_lR2ZFcryNFiHkS
      w129EghGpwkpxaTn_THJTCglNbADko1MZBCdwzJxwqZc-1RlpO2HibUY
      yXSwO97BSe0_evZKdjvvKSgsIqjytKSeAMbhMBdMma622_BG5t4sdbuC
      HtFjp9iJmkio47AIwqkZV1aIZsv33uPUqBBCXbYoQJwt7mxPftHmNlGo
      OSMxR_3thmXTCm4US-xiNOyhbm8afKK64jU6_TPtQHiJeQJxz9G3Tx-0
      83B745_AfYOnlC9w"
}
```

Figure 61: Signature #1 JSON

4.8.3. Second Signing Operation

The following are generated before completing the second signing operation:

- JWS Unprotected Header; this example uses the header from Figure 62.

```
{
  "alg": "ES512",
  "kid": "bilbo.baggins@hobbiton.example"
}
```

Figure 62: Signature #2 JWS Unprotected Header JSON

The empty string (as there is no JWS Protected Header) and Payload content (Figure 8) are combined as described in [I-D.ietf-jose-json-web-signature] to produce the JWS Signing Input Figure 63.

```
.
SXTigJlzIGEgZGFuZ2Vyb3VzIGJ1c2luZXNzLCBGcm9kbywgZ29pbmcgb3V0IH
lvdXIgZG9vci4gWW91IHN0ZXAgb250byB0aGUgcm9hZCwgYW5kIGlmIHlvdSBk
b24ndCBrZWVwIHlvdXIgZmVldCwgdGhlcmXigJlzIG5vIGtub3dpbmcgd2hlcm
UgeW91IG1pZ2h0IGJlIHN3ZXB0IG9mZiB0by4
```

Figure 63: JWS Signing Input

Performing the signature operation over the JWS Signing Input (Figure 63) produces the JWS Signature (Figure 64).

```
ARcVLnaJJaUWG8fG-8t5BREVAuTY8n8YHjwDO1muhcdCoFZFFjfISu0Cdkn9Yb
dlmi54ho0x924DUz8sK7ZXkhc7AFM8ObLfTvNCrqcI3Jkl2U5IX3utNhODH6v7
xgy1Qahsn0fyb4zSAkje8bAWz4vIfj5pCMYxxm4fgV3q7ZYhm5eD
```

Figure 64: JWS Signature #2, base64url-encoded

The following is the assembled second signature serialized as JSON:

```
{
  "header": {
    "alg": "ES512",
    "kid": "bilbo.baggins@hobbiton.example"
  },
  "signature": "ARcVLnaJJaUWG8fG-8t5BREVAuTY8n8YHjwDO1muhcdCoF
      ZFFjfISu0Cdkn9Ybdlmi54ho0x924DUz8sK7ZXkhc7AFM8ObLfTvNCrq
      cI3Jkl2U5IX3utNhODH6v7xgy1Qahsn0fyb4zSAkje8bAWz4vIfj5pCM
      Yxxm4fgV3q7ZYhm5eD"
}
```

Figure 65: Signature #2 JSON

4.8.4. Third Signing Operation

The following are generated before completing the third signing operation:

- JWS Protected Header; this example uses the header from Figure 66, encoded using [RFC4648] base64url to produce Figure 67.

```
{
  "alg": "HS256",
  "kid": "018c0ae5-4d9b-471b-bfd6-eef314bc7037"
}
```

Figure 66: Signature #3 JWS Protected Header JSON

```
eyJhbGciOiJIUzI1NiIsImtpZCI6IjAxOGMwYWU1LTRkOWItNDcxYi1iZmQ2LW
VlZjMxNGJjNzAzNyJ9
```

Figure 67: Signature #3 JWS Protected Header, base64url-encoded

The JWS Protected Header (Figure 67) and Payload content (Figure 8) are combined as described in [I-D.ietf-jose-json-web-signature] to produce the JWS Signing Input Figure 68.

```
eyJhbGciOiJIUzI1NiIsImtpZCI6IjAxOGMwYWU1LTRkOWItNDcxYi1iZmQ2LW
VlZjMxNGJjNzAzNyJ9
.
SXTigJlzIGEgZGFuZ2Vyb3VzIGJ1c2luZXNzLCBGcm9kbywgZ29pbmcgb3V0IH
lvdXIgZG9vci4gWW91IHN0ZXAgb250byB0aGUgcm9hZCwgYW5kIGlmIHlvdSBk
b24ndCBrZWVwIHlvdXIgZmVldCwgdGhlcmXigJlzIG5vIGtub3dpbmcgd2hlcm
UgeW91IG1pZ2h0IGJlIHN3ZXB0IG9mZiB0by4
```

Figure 68: JWS Signing Input

Performing the signature operation over the JWS Signing Input (Figure 68) produces the JWS Signature (Figure 69).

```
s0h6KThzkfBBBkLspW1h84VsJZFTsPPqMDA7g1Md7p0
```

Figure 69: JWS Signature #3, base64url-encoded

The following is the assembled third signature serialized as JSON:

```
{
  "protected": "eyJhbGciOiJIUzI1NiIsImtpZCI6IjAxOGMwYWU1LTRkOW
      ItNDcxYi1iZmQ2LWVlZjMxNGJjNzAzNyJ9",
  "signature": "s0h6KThzkfBBBkLspW1h84VsJZFTsPPqMDA7g1Md7p0"
}
```

Figure 70: Signature #3 JSON

4.8.5. Output Results

The following compose the resulting JWS object:

- Payload content (Figure 8)
- Signature #1 JSON (Figure 61)
- Signature #2 JSON (Figure 65)
- Signature #3 JSON (Figure 70)

The Compact Serialization is not presented because it does not support this use case; the JSON Flattened Serialization is not presented because there is more than one signature.

The resulting JWS object using the JSON General Serialization:

```
{
  "payload": "SXTigJlzIGEgZGFuZ2Vyb3VzIGJ1c2luZXNzLCBGcm9kbywg
      Z29pbmcgb3V0IHlvdXIgZG9vci4gWW91IHN0ZXAgb250byB0aGUgcm9h
      ZCwgYW5kIGlmIHlvdSBkb24ndCBrZWVwIHlvdXIgZmVldCwgdGhlcmXi
      gJlzIG5vIGtub3dpbmcgd2hlcmUgeW91IG1pZ2h0IGJlIHN3ZXB0IG9m
      ZiB0by4",
  "signatures": [
    {
      "protected": "eyJhbGciOiJSUzI1NiJ9",
      "header": {
        "kid": "bilbo.baggins@hobbiton.example"
      },
      "signature": "MIsjqtVlOpa71KE-Mss8_Nq2YH4FGhiocsqrgi5Nvy
          G53uoimic1tcMdSg-qptrzZc7CG6Svw2Y13TDIqHzTUrL_lR2ZFc
          ryNFiHkSw129EghGpwkpxaTn_THJTCglNbADko1MZBCdwzJxwqZc
          -1RlpO2HibUYyXSwO97BSe0_evZKdjvvKSgsIqjytKSeAMbhMBdM
          ma622_BG5t4sdbuCHtFjp9iJmkio47AIwqkZV1aIZsv33uPUqBBC
          XbYoQJwt7mxPftHmNlGoOSMxR_3thmXTCm4US-xiNOyhbm8afKK6
          4jU6_TPtQHiJeQJxz9G3Tx-083B745_AfYOnlC9w"
    },
    {
      "header": {
        "alg": "ES512",
        "kid": "bilbo.baggins@hobbiton.example"
      },
      "signature": "ARcVLnaJJaUWG8fG-8t5BREVAuTY8n8YHjwDO1muhc
          dCoFZFFjfISu0Cdkn9Ybdlmi54ho0x924DUz8sK7ZXkhc7AFM8Ob
          LfTvNCrqcI3Jkl2U5IX3utNhODH6v7xgy1Qahsn0fyb4zSAkje8b
          AWz4vIfj5pCMYxxm4fgV3q7ZYhm5eD"
    },
    {
      "protected": "eyJhbGciOiJIUzI1NiIsImtpZCI6IjAxOGMwYWU1LT
          RkOWItNDcxYi1iZmQ2LWVlZjMxNGJjNzAzNyJ9",
      "signature": "s0h6KThzkfBBBkLspW1h84VsJZFTsPPqMDA7g1Md7p
          0"
    }
  ]
}
```


Figure 71: JSON General Serialization

5. JSON Web Encryption Examples

The following sections demonstrate how to generate various JWE objects.

All of the succeeding examples (unless otherwise noted) use the following plaintext content (an abridged quote from "The Fellowship

of the Ring" [LOTR-FELLOWSHIP]), serialized as UTF-8. The sequence "\xe2\x80\x93" is substituted for (U+2013 EN DASH), and line breaks (U+000A LINE FEED) replace some " " (U+0020 SPACE) characters to improve formatting:

```
You can trust us to stick with you through thick and
thin\xe2\x80\x93to the bitter end. And you can trust us to
keep any secret of yours\xe2\x80\x93closer than you keep it
yourself. But you cannot trust us to let you face trouble
alone, and go off without a word. We are your friends, Frodo.
```

Figure 72: Plaintext content

5.1. Key Encryption using RSA v1.5 and AES-HMAC-SHA2

This example illustrates encrypting content using the "RSA1_5" (RSAES-PKCS1-v1_5) key encryption algorithm and the "A128CBC-HS256" (AES-128-CBC-HMAC-SHA-256) content encryption algorithm.

Note that RSAES-PKCS1-v1_5 uses random data to generate the ciphertext; it might not be possible to exactly replicate the results in this section.

Note that only the RSA public key is necessary to perform the encryption. However, the example includes the RSA private key to allow readers to validate the output.

Note that whitespace is added for readability as described in Section 1.1.

5.1.1. Input Factors

The following are supplied before beginning the encryption process:

- Plaintext content; this example uses the content from Figure 72.
- RSA public key; this example uses the key from Figure 73.
- "alg" parameter of "RSA1_5".
- "enc" parameter of "A128CBC-HS256".

```
{
  "kty": "RSA",
  "kid": "frodo.baggins@hobbiton.example",
  "use": "enc",
  "n": "maxhbsmBtdQ3CNrKvprUE6n9lYcregDMLYNeTAWcLj8NnPU9XIYegT
      HVHQjxKDSHP2l-F5jS7sppG1wgdAqZyhnWvXhYNvcM7RfgKxqNx_xAHx
      6f3yy7s-M9PSNCwPC2lh6UAkR4I00EhV9lrypM9Pi4lBUop9t5fS9W5U
      NwaAllhrd-osQGPjIeI1deHTwx-ZTHu3C60Pu_LJIl6hKn9wbwaUmA4c
      R5Bd2pgbaY7ASgsjCUbtYJaNIHSoHXprUdJZKUMAzV0WOKPfA6OPI4oy
      pBadjvMZ4ZAj3BnXaSYsEZhaueTXvZB4eZOAjIyh2e_VOIKVMsnDrJYA
      VotGlvMQ",
  "e": "AQAB",
  "d": "Kn9tgoHfiTVi8uPu5b9TnwyHwG5dK6RE0uFdlpCGnJN7ZEi963R7wy
      bQ1PLAHmpIbNTztfrheoAniRV1NCIqXaW_qS461xiDTp4ntEPnqcKsyO
      5jMAji7-CL8vhpYYowNFvIesgMoVaPRYMYT9TW63hNM0aWs7USZ_hLg6
      Oe1mY0vHTI3FucjSM86Nff4oIENt43r2fspgEPGRrdE6fpLc9Oaq-qeP
      1GFULimrRdndm-P8q8kvN3KHlNAtEgrQAgTTgz80S-3VD0FgWfgnb1PN
      miuPUxO8OpI9KDIfu_acc6fg14nsNaJqXe6RESvhGPH2afjHqSy_Fd2v
      pzj85bQQ",
  "p": "2DwQmZ43FoTnQ8IkUj3BmKRf5Eh2mizZA5xEJ2MinUE3sdTYKSLtaE
      oekX9vbBZuWxHdVhM6UnKCJ_2iNk8Z0ayLYHL0_G21aXf9-unynEpUsH
      7HHTklLpYAzOOx1ZgVljoxAdWNn3hiEFrjZLZGS7lOH-a3QQlDDQoJOJ
      2VFmU",
  "q": "te8LY4-W7IyaqH1ExujjMqkTAlTeRbv0VLQnfLY2xINnrWdwiQ93_V
      F099aP1ESeLja2nw-6iKIe-qT7mtCPozKfVtUYfz5HrJ_XY2kfexJINb
      9lhZHMv5p1skZpeIS-GPHCC6gRlKo1q-idn_qxyusfWv7WAxlSVfQfk8
      d6Et0",
  "dp": "UfYKcL_or492vVc0PzwLSplbg4L3-Z5wL48mwiswbpzOyIgd2xHTH
      QmjJpFAIZ8q-zf9RmgJXkDrFs9rkdxPtAsL1WYdeCT5c125Fkdg317JV
      RDo1inX7x2Kdh8ERCreW8_4zXItuTl_KiXZNU5lvMQjWbIw2eTx1lpsf
      lo0rYU",
  "dq": "iEgcO-QfpepdH8FWd7mUFyrXdnOkXJBCogChY6YKuIHGc_p8Le9Mb
      pFKESzEaLlN1Ehf3B6oGBl5Iz_ayUlZj2IoQZ82znoUrpa9fVYNot87A
      CfzIG7q9Mv7RiPAderZi03tkVXAdaBau_9vs5rS-7HMtxkVrxSUvJY14
      TkXlHE",
  "qi": "kC-lzZOqoFaZCr5l0tOVtREKoVqaAYhQiqIRGL-MzS4sCmRkxm5vZ
      lXYx6RtE1n_Aagjqajlkjie GlxTTThHD8Iga6foGBMaAr5uR1hGQpSc7
      Gl7CF1DZkBJMTQN6EshYzZfxW08mIO8M6Rzuh0beL6fG9mkDcIyPrBXx
      2bQ_mM"
}
```


Figure 73: RSA 2048-bit Key, in JWK format

(*NOTE*: While the key includes the private parameters, only the public parameters "e" and "n" are necessary for the encryption operation.)

5.1.2. Generated Factors

The following are generated before encrypting:

- AES symmetric key as the Content Encryption Key (CEK); this example uses the key from Figure 74
- Initialization vector/nonce; this example uses the initialization vector from Figure 75

```
3qyTVhIWt5juqZUCpfRqpvauwB956MEJL2Rt-8qXKSo
```

Figure 74: Content Encryption Key, base64url-encoded

```
bbd5sTkYwhAIqfHsx8DayA
```

Figure 75: Initialization Vector, base64url-encoded

5.1.3. Encrypting the Key

Performing the key encryption operation over the CEK (Figure 74) with the RSA key (Figure 73) results in the following encrypted key:

```
laLxI0j-nLH-_BgLOXMozKxmy9gffy2gTdvqzfTihJBuuzxg0V7yk1WClnQePF
vG2K-pvSlWc9BRIazDrn50RcRai__3TDON395H3c62tIouJJ4XaRvYHFjZTZ2G
Xfz8YAImcc91Tfk0WXC2F5Xbb71ClQ1DDH151tlpH77f2ff7xiSxh9oSewYrcG
TSLUeeCt36r1Kt3OSj7EyBQXoZlN7IxbyhMAfgIe7Mv1rOTOI5I8NQqeXXW8Vl
zNmoxaGMny3YnGir5Wf6Qt2nBq4qDaPdnaAuuGUGEecelIO1wx1BpyIfgvfjOh
MBs9M8XL223Fg47xlGsMXdfuY-4jaqVw
```

Figure 76: Encrypted Key, base64url-encoded

5.1.4. Encrypting the Content

The following are generated before encrypting the plaintext:

- JWE Protected Header; this example uses the header from Figure 77, encoded using [RFC4648] base64url to produce Figure 78.

```
{
  "alg": "RSA1_5",
  "kid": "frodo.baggins@hobbiton.example",
  "enc": "A128CBC-HS256"
}
```

Figure 77: JWE Protected Header JSON

```
eyJhbGciOiJSU0ExXzUiLCJraWQiOiJmcm9kby5iYWdnaW5zQGhvYmJpdG9uLm
V4YW1wbGUiLCJlbmMiOiJBMTI4Q0JDLUhTMjU2In0
```

Figure 78: JWE Protected Header, base64url-encoded

Performing the content encryption operation on the Plaintext (Figure 72) using the following:

- CEK (Figure 74);
- Initialization vector/nonce (Figure 75); and
- JWE Protected Header (Figure 77) as authenticated data

produces the following:

- Ciphertext from Figure 79.
- Authentication tag from Figure 80.

```
0fys_TY_na7f8dwSfXLiYdHaA2DxUjD67ieF7fcVbIR62JhJvGZ4_FNVSiGc_r
aa0HnLQ6s1P2sv3Xzl1p1l_o5wR_RsSzrS8Z-wnI3Jvo0mkpEEnlDmZvDu_k8O
WzJv7eZVEqiWKdyVzFhPpiyQU28GLOpRc2VbVbK4dQKPdNTjPPEmRqcaGeTWZV
yeSUvf5k59yJZxRuSvWFf6KrNtmRdZ8R4mDOjHSrM_s8uwIFcqt4r5GX8TKaI0
zT5CbL5Qlw3sRc7u_hg0yKVOiRytEAEs3vZkcfLkP6nbXdC_PkMdNS-ohP78T2
O6_7uInMGhFeX4ctHG7VelHGiT93JfWDEQi5_V9UN1rhXNrYu-0fVMkZAKX3VW
i7lzA6BP430m
```

Figure 79: Ciphertext, base64url-encoded

```
kvKuFBXHe5mQr4lqgobAUg
```

Figure 80: Authentication Tag, base64url-encoded

5.1.5. Output Results

The following compose the resulting JWE object:

- JWE Protected Header (Figure 78).
- Encrypted Key (Figure 76).
- Initialization vector/nonce (Figure 75).
- Ciphertext (Figure 79).
- Authentication Tag (Figure 80).

The resulting JWE object using the Compact serialization:

```
eyJhbGciOiJSU0ExXzUiLCJraWQiOiJmcm9kby5iYWdnaW5zQGhvYmJpdG9uLm
V4YW1wbGUiLCJlbmMiOiJBMTI4Q0JDLUhTMjU2In0
.
laLxI0j-nLH-_BgLOXMozKxmy9gffy2gTdvqzfTihJBuuzxg0V7yk1WClnQePF
vG2K-pvSlWc9BRIazDrn50RcRai__3TDON395H3c62tIouJJ4XaRvYHFjZTZ2G
Xfz8YAImcc91Tfk0WXC2F5Xbb71ClQ1DDH151tlpH77f2ff7xiSxh9oSewYrcG
TSLUeeCt36r1Kt3OSj7EyBQXoZlN7IxbyhMAfgIe7Mv1rOTOI5I8NQqeXXW8Vl
zNmoxaGMny3YnGir5Wf6Qt2nBq4qDaPdnaAuuGUGEecelIO1wx1BpyIfgvfjOh
MBs9M8XL223Fg47xlGsMXdfuY-4jaqVw
.
bbd5sTkYwhAIqfHsx8DayA
.
0fys_TY_na7f8dwSfXLiYdHaA2DxUjD67ieF7fcVbIR62JhJvGZ4_FNVSiGc_r
aa0HnLQ6s1P2sv3Xzl1p1l_o5wR_RsSzrS8Z-wnI3Jvo0mkpEEnlDmZvDu_k8O
WzJv7eZVEqiWKdyVzFhPpiyQU28GLOpRc2VbVbK4dQKPdNTjPPEmRqcaGeTWZV
yeSUvf5k59yJZxRuSvWFf6KrNtmRdZ8R4mDOjHSrM_s8uwIFcqt4r5GX8TKaI0
zT5CbL5Qlw3sRc7u_hg0yKVOiRytEAEs3vZkcfLkP6nbXdC_PkMdNS-ohP78T2
O6_7uInMGhFeX4ctHG7VelHGiT93JfWDEQi5_V9UN1rhXNrYu-0fVMkZAKX3VW
i7lzA6BP430m
.
kvKuFBXHe5mQr4lqgobAUg
```

Figure 81: Compact Serialization

The resulting JWE object using the JSON General Serialization:

```
{
  "recipients": [
    {
      "encrypted_key": "laLxI0j-nLH-_BgLOXMozKxmy9gffy2gTdvqzf
          TihJBuuzxg0V7yk1WClnQePFvG2K-pvSlWc9BRIazDrn50RcRai_
          _3TDON395H3c62tIouJJ4XaRvYHFjZTZ2GXfz8YAImcc91Tfk0WX
          C2F5Xbb71ClQ1DDH151tlpH77f2ff7xiSxh9oSewYrcGTSLUeeCt
          36r1Kt3OSj7EyBQXoZlN7IxbyhMAfgIe7Mv1rOTOI5I8NQqeXXW8
          VlzNmoxaGMny3YnGir5Wf6Qt2nBq4qDaPdnaAuuGUGEecelIO1wx
          1BpyIfgvfjOhMBs9M8XL223Fg47xlGsMXdfuY-4jaqVw"
    }
  ],
  "protected": "eyJhbGciOiJSU0ExXzUiLCJraWQiOiJmcm9kby5iYWdnaW
      5zQGhvYmJpdG9uLmV4YW1wbGUiLCJlbmMiOiJBMTI4Q0JDLUhTMjU2In
      0",
  "iv": "bbd5sTkYwhAIqfHsx8DayA",
  "ciphertext": "0fys_TY_na7f8dwSfXLiYdHaA2DxUjD67ieF7fcVbIR62
      JhJvGZ4_FNVSiGc_raa0HnLQ6s1P2sv3Xzl1p1l_o5wR_RsSzrS8Z-wn
      I3Jvo0mkpEEnlDmZvDu_k8OWzJv7eZVEqiWKdyVzFhPpiyQU28GLOpRc
      2VbVbK4dQKPdNTjPPEmRqcaGeTWZVyeSUvf5k59yJZxRuSvWFf6KrNtm
      RdZ8R4mDOjHSrM_s8uwIFcqt4r5GX8TKaI0zT5CbL5Qlw3sRc7u_hg0y
      KVOiRytEAEs3vZkcfLkP6nbXdC_PkMdNS-ohP78T2O6_7uInMGhFeX4c
      tHG7VelHGiT93JfWDEQi5_V9UN1rhXNrYu-0fVMkZAKX3VWi7lzA6BP4
      30m",
  "tag": "kvKuFBXHe5mQr4lqgobAUg"
}
```


Figure 82: JSON General Serialization

The resulting JWE object using the JSON Flattened Serialization:

```
{
  "protected": "eyJhbGciOiJSU0ExXzUiLCJraWQiOiJmcm9kby5iYWdnaW
      5zQGhvYmJpdG9uLmV4YW1wbGUiLCJlbmMiOiJBMTI4Q0JDLUhTMjU2In
      0",
  "encrypted_key": "laLxI0j-nLH-_BgLOXMozKxmy9gffy2gTdvqzfTihJ
      Buuzxg0V7yk1WClnQePFvG2K-pvSlWc9BRIazDrn50RcRai__3TDON39
      5H3c62tIouJJ4XaRvYHFjZTZ2GXfz8YAImcc91Tfk0WXC2F5Xbb71ClQ
      1DDH151tlpH77f2ff7xiSxh9oSewYrcGTSLUeeCt36r1Kt3OSj7EyBQX
      oZlN7IxbyhMAfgIe7Mv1rOTOI5I8NQqeXXW8VlzNmoxaGMny3YnGir5W
      f6Qt2nBq4qDaPdnaAuuGUGEecelIO1wx1BpyIfgvfjOhMBs9M8XL223F
      g47xlGsMXdfuY-4jaqVw",
  "iv": "bbd5sTkYwhAIqfHsx8DayA",
  "ciphertext": "0fys_TY_na7f8dwSfXLiYdHaA2DxUjD67ieF7fcVbIR62
      JhJvGZ4_FNVSiGc_raa0HnLQ6s1P2sv3Xzl1p1l_o5wR_RsSzrS8Z-wn
      I3Jvo0mkpEEnlDmZvDu_k8OWzJv7eZVEqiWKdyVzFhPpiyQU28GLOpRc
      2VbVbK4dQKPdNTjPPEmRqcaGeTWZVyeSUvf5k59yJZxRuSvWFf6KrNtm
      RdZ8R4mDOjHSrM_s8uwIFcqt4r5GX8TKaI0zT5CbL5Qlw3sRc7u_hg0y
      KVOiRytEAEs3vZkcfLkP6nbXdC_PkMdNS-ohP78T2O6_7uInMGhFeX4c
      tHG7VelHGiT93JfWDEQi5_V9UN1rhXNrYu-0fVMkZAKX3VWi7lzA6BP4
      30m",
  "tag": "kvKuFBXHe5mQr4lqgobAUg"
}
```


Figure 83: JSON Flattened Serialization

5.2. Key Encryption using RSA-OAEP with AES-GCM

This example illustrates encrypting content using the "RSA-OAEP" (RSAES-OAEP) key encryption algorithm and the "A256GCM" (AES-GCM) content encryption algorithm.

Note that RSAES-OAEP uses random data to generate the ciphertext; it might not be possible to exactly replicate the results in this section.

Note that only the RSA public key is necessary to perform the encryption. However, the example includes the RSA private key to allow readers to validate the output.

Note that whitespace is added for readability as described in Section 1.1.

5.2.1. Input Factors

The following are supplied before beginning the encryption process:

- Plaintext content; this example uses the plaintext from Figure 72.

- RSA public key; this example uses the key from Figure 84.

- "alg" parameter of "RSA-OAEP"

- "enc" parameter of "A256GCM"


```
{
  "kty": "RSA",
  "kid": "samwise.gamgee@hobbiton.example",
  "use": "enc",
  "n": "wbdxI55VaanZXPY29Lg5hdmv2XhvqAhoxUkanfzf2-5zVUxa6prHRr
      I4pP1AhoqJRlZfYtWWd5mmHRG2pAHIlh0ySJ9wi0BioZBl1XP2e-C-Fy
      XJGcTy0HdKQWlrfhTm42EW7Vv04r4gfao6uxjLGwfpGrZLarohiWCPnk
      Nrg71S2CuNZSQBIPGjXfkmIy2tl_VWgGnL22GplyXj5YlBLdxXp3XeSt
      sqo571utNfoUTU8E4qdzJ3U1DItoVkPGsMwlmmnJiwA7sXRItBCivR4M
      5qnZtdw-7v4WuR4779ubDuJ5nalMv2S66-RPcnFAzWSKxtBDnFJJDGIU
      e7Tzizjg1nms0Xq_yPub_UOlWn0ec85FCft1hACpWG8schrOBeNqHBOD
      FskYpUc2LC5JA2TaPF2dA67dg1TTsC_FupfQ2kNGcE1LgprxKHcVWYQb
      86B-HozjHZcqtauBzFNV5tbTuB-TpkcvJfNcFLlH3b8mb-H_ox35FjqB
      SAjLKyoeqfKTpVjvXhd09knwgJf6VKq6UC418_TOljMVfFTWXUxlnfhO
      OnzW6HSSzD1c9WrCuVzsUMv54szidQ9wf1cYWf3g5qFDxDQKis99gcDa
      iCAwM3yEBIzuNeeCa5dartHDb1xEB_HcHSeYbghbMjGfasvKn0aZRsnT
      yC0xhWBlsolZE",
  "e": "AQAB",
  "alg": "RSA-OAEP",
  "d": "n7fzJc3_WG59VEOBTkayzuSMM780OJQuZjN_KbH8lOZG25ZoA7T4Bx
      cc0xQn5oZE5uSCIwg91oCt0JvxPcpmqzaJZg1nirjcWZ-oBtVk7gCAWq
      -B3qhfF3izlbkosrzjHajIcY33HBhsy4_WerrXg4MDNE4HYojy68TcxT
      2LYQRxUOCf5TtJXvM8olexlSGtVnQnDRutxEUCwiewfmmrfveEogLx9E
      A-KMgAjTiISXxqIXQhWUQX1G7v_mV_Hr2YuImYcNcHkRvp9E7ook0876
      DhkO8v4UOZLwA1OlUX98mkoqwc58A_Y2lBYbVx1_s5lpPsEqbbH-nqIj
      h1fL0gdNfihLxnclWtW7pCztLnImZAyeCWAG7ZIfv-Rn9fLIv9jZ6r7r
      -MSH9sqbuziHN2grGjD_jfRluMHa0l84fFKl6bcqN1JWxPVhzNZo01yD
      F-1LiQnqUYSepPf6X3a2SOdkqBRiquE6EvLuSYIDpJq3jDIsgoL8Mo1L
      oomgiJxUwL_GWEOGu28gplyzm-9Q0U0nyhEf1uhSR8aJAQWAiFImWH5W
      _IQT9I7-yrindr_2fWQ_i1UgMsGzA7aOGzZfPljRy6z-tY_KuBG00-28
      S_aWvjyUc-Alp8AUyKjBZ-7CWH32fGWK48j1t-zomrwjL_mnhsPbGs0c
      9WsWgRzI-K8gE",
  "p": "7_2v3OQZzlPFcHyYfLABQ3XP85Es4hCdwCkbDeltaUXgVy9l9etKgh
      vM4hRkOvbb01kYVuLFmxIkCDtpi-zLCYAdXKrAK3PtSbtzld_XZ9nlsY
      a_QZWpXB_IrtFjVfdKUdMz94pHUhFGFj7nr6NNxfpiHSHWFE1zD_AC3m
      Y46J961Y2LRnreVwAGNw53p07Db8yD_92pDa97vqcZOdgtybH9q6uma-
      RFNhO1AoiJhYZj69hjmMRXx-x56HO9cnXNbmzNSCFCKnQmn4GQLmRj9s
      fbZRqL94bbtE4_e0Zrpo8RNo8vxRLqQNwIy85fc6BRgBJomt8QdQvIgP
      gWCv5HoQ",
  "q": "zqOHk1P6WN_rHuM7ZF1cXH0x6RuOHq67WuHiSknqQeefGBA9PWs6Zy
      KQCO-O6mKXtcgE8_Q_hA2kMRcKOcvHil1hqMCNSXlflM7WPRPZu2qCDc
      qssd_uMbP-DqYthH_EzwL9KnYoH7JQFxxmcv5An8oXUtTwk4knKjkIYG
```

```
        RuUwfQTus0w1NfjFAyxOOiAQ37ussIcE6C6ZSsM3n41UlbJ7TCqewzVJ
        aPJN5cxjySPZPD3Vp01a9YgAD6a3IIaKJdIxJS1ImnfPevSJQBE79-EX
        e2kSwVgOzvt-gsmM29QQ8veHy4uAqca5dZzMs7hkkHtw1z0jHV90epQJ
        JlXXnH8Q",
    "dp": "19oDkBh1AXelMIxQFm2zZTqUhAzCIr4xNIGEPNoDt1jK83_FJA-xn
        x5kA7-1erdHdms_Ef67HsONNv5A60JaR7w8LHnDiBGnjdaUmmuO8XAxQ
        J_ia5mxjxNjS6E2yD44USo2JmHvzeeNczq25elqbTPLhUpGo1IZuG72F
        ZQ5gTjXoTXC2-xtCDEUZfaUNh4IeAipfLugbpe0JAFlFfrTDAMUFpC3i
        XjxqzbEanflwPvj6V9iDSgjj8SozSM0dLtxvu0LIeIQAeEgT_yXcrKGm
        pKdSO08kLBx8VUjkbv_3Pn20Gyu2YEuwpFlM_H1NikuxJNKFGmnAq9Lc
        nwwT0jvoQ",
    "dq": "S6p59KrlmzGzaQYQM3o0XfHCGvfqHLYjCO557HYQf72O9kLMCfd_1
        VBEqeD-1jjwELKDjck8kOBl5UvohK1oDfSP1DleAy-cnmL29DqWmhgwM
        1ip0CCNmkmsmDSlqkUXDi6sAaZuntyukyflI-qSQ3C_BafPyFaKrt1fg
        dyEwYa08pESKwwWisy7KnmoUvaJ3SaHmohFS78TJ25cfc10wZ9hQNOrI
        ChZlkiOdFCtxDqdmCqNacnhgE3bZQjGp3n83ODSz9zwJcSUvODlXBPc2
        AycH6Ci5yjbxt4Ppox_5pjm6xnQkiPgj01GpsUssMmBN7iHVsrE7N2iz
        nBNCeOUIQ",
    "qi": "FZhClBMywVVjnuUud-05qd5CYU0dK79akAgy9oX6RX6I3IIIPckCc
        iRrokxglZn-omAY5CnCe4KdrnjFOT5YUZE7G_Pg44XgCXaarLQf4hl80
        oPEf6-jJ5Iy6wPRx7G2e8qLxnh9cOdf-kRqgOS3F48Ucvw3ma5V6KGMw
        QqWFeV31XtZ8l5cVI-I3NzBS7qltpUVgz2Ju021eyc7IlqgzR98qKONl
        27DuEES0aK0WE97jnsyO27Yp88Wa2RiBrEocM89QZI1seJiGDizHRUP4
        UZxw9zsXww46wy0P6f9grnYp7t8LkyDDk8eoI4KX6SNMNVcyVS9IWjlq
        8EzqZEKIA"
}
```


Figure 84: RSA 4096-bit Key

(*NOTE*: While the key includes the private parameters, only the public parameters "e" and "n" are necessary for the encryption operation.)

5.2.2. Generated Factors

The following are generated before encrypting:

- AES symmetric key as the Content Encryption Key (CEK); this example uses the key from Figure 85.

- Initialization vector/nonce; this example uses the initialization vector/nonce from Figure 86.


```
mYMfsggkTAm0TbvtlFh2hyoXnbEzJQjMxmgLN3d8xXA
```


Figure 85: Content Encryption Key, base64url-encoded

```
-nBoKLH0YkLZPSI9
```

Figure 86: Initialization Vector, base64url-encoded

5.2.3. Encrypting the Key

Performing the key encryption operation over the CEK (Figure 85)) with the RSA key (Figure 84) produces the following encrypted key:

```
rT99rwrBTbTI7IJM8fU3Eli7226HEB7IchCxNuh7lCiud48LxeolRdtFF4nzQi
beYOl5S_PJsAXZwSXtDePz9hk-BbtsTBqC2UsPOdwjC9NhNupNNu9uHIVftDyu
cvI6hvALeZ6OGnhNV4v1zx2k7O1D89mAzfw-_kT3tkuorpDU-CpBENfIHX1Q58
-Aad3FzMuo3Fn9buEP2yXakLXYa15BUXQsupM4A1GD4_H4Bd7V3u9h8Gkg8Bpx
KdUV9ScfJQTcYm6eJEBz3aSwIaK4T3-dwWpuBOhROQXBosJzS1asnuHtVMt2pK
IIfux5BC6huIvmY7kzV7W7aIUrpYm_3H4zYvyMeq5pGqFmW2k8zpO878TRlZx7
pZfPYDSXZyS0CfKKkMozT_qiCwZTSz4duYnt8hS4Z9sGthXn9uDqd6wycMagnQ
fOTs_lycTWmY-aqWVDKhjYNRf03NiwRtb5BE-tOdFwCASQj3uuAgPGrO2AWBe3
8UjQb0lvXn1SpyvYZ3WFc7WOJYaTa7A8DRn6MC6T-xDmMuxC0G7S2rscw5lQQU
06MvZTlFOt0UvfuKBa03cxA_nIBIhLMjY2kOTxQMmpDPTr6Cbo8aKaOnx6ASE5
Jx9paBpnNmOOKH35j_QlrQhDWUN6A2Gg8iFayJ69xDEdHAVCGRzN3woEI2ozDR
s
```

Figure 87: Encrypted Key, base64url-encoded

5.2.4. Encrypting the Content

The following are generated before encrypting the plaintext:

- JWE Protected Header; this example uses the the header from Figure 88, encoded using [RFC4648] base64url to produce Figure 89.

```
{
  "alg": "RSA-OAEP",
  "kid": "samwise.gamgee@hobbiton.example",
  "enc": "A256GCM"
}
```

Figure 88: JWE Protected Header JSON

```
eyJhbGciOiJSU0EtT0FFUCIsImtpZCI6InNhbXdpc2UuZ2FtZ2VlQGhvYmJpdG
9uLmV4YW1wbGUiLCJlbmMiOiJBMjU2R0NNIn0
```

Figure 89: JWE Protected Header, base64url-encoded

Performing the content encryption operation over the Plaintext (Figure 72) with the following:

- CEK (Figure 85);

- Initialization vector/nonce (Figure 86); and
- JWE Protected Header (Figure 89) as authenticated data

produces the following:

- Ciphertext from Figure 90.
- Authentication tag from Figure 91.

o4k2cnGN8rSSw3IDo1YuySkqeS_t2m1GXklSgqBdpACm6UJuJowOHC5ytjqYgR
L-I-soPlwqMUf4UgRWWeaOGNw6vGW-xyM01lTYxrXfVzIIaRdhYtEMRBvBWbEw
P7ua1DRfvaOjgZv6Ifa3brcAM64d8p5lhhNcizPersuhw5f-pGYzseva-TUaL8
iWnctc-sSwy7SQmRkfhDjwbz0fz6kFovEgj64X1I5s7E6GLp5fnbYGLa1QUiML
7Cc2GxgvI7zqWo0YIEc7aCflLG1-8BboVWFdZKLK9vNoycrYHumwzKluLWEbSV
maPpOslY2n525DxDfWaVFUfKQxMF56vn4B9QMpWAbnypNimbM8zVOw

Figure 90: Ciphertext, base64url-encoded

UCGiqJxhBI3IFVdPalHHvA

Figure 91: Authentication Tag, base64url-encoded

5.2.5. Output Results

The following compose the resulting JWE object:

- JWE Protected Header (Figure 89)
- Encrypted key (Figure 87)
- Initialization vector/nonce (Figure 86)
- Ciphertext (Figure 90)
- Authentication tag (Figure 91)

The resulting JWE object using the Compact serialization:

```
eyJhbGciOiJSU0EtT0FFUCIsImtpZCI6InNhbXdpc2UuZ2FtZ2VlQGhvYmJpdG
9uLmV4YW1wbGUiLCJlbmMiOiJBMjU2R0NNIn0
.
rT99rwrBTbTI7IJM8fU3Eli7226HEB7IchCxNuh7lCiud48LxeolRdtFF4nzQi
beYOl5S_PJsAXZwSXtDePz9hk-BbtsTBqC2UsPOdwjC9NhNupNNu9uHIVftDyu
cvI6hvALeZ6OGnhNV4v1zx2k7O1D89mAzfw-_kT3tkuorpDU-CpBENfIHX1Q58
-Aad3FzMuo3Fn9buEP2yXakLXYa15BUXQsupM4A1GD4_H4Bd7V3u9h8Gkg8Bpx
KdUV9ScfJQTcYm6eJEBz3aSwIaK4T3-dwWpuBOhROQXBosJzS1asnuHtVMt2pK
IIfux5BC6huIvmY7kzV7W7aIUrpYm_3H4zYvyMeq5pGqFmW2k8zpO878TRlZx7
pZfPYDSXZyS0CfKKkMozT_qiCwZTSz4duYnt8hS4Z9sGthXn9uDqd6wycMagnQ
fOTs_lycTWmY-aqWVDKhjYNRf03NiwRtb5BE-tOdFwCASQj3uuAgPGrO2AWBe3
8UjQb0lvXn1SpyvYZ3WFc7WOJYaTa7A8DRn6MC6T-xDmMuxC0G7S2rscw5lQQU
06MvZTlFOt0UvfuKBa03cxA_nIBIhLMjY2kOTxQMmpDPTr6Cbo8aKaOnx6ASE5
Jx9paBpnNmOOKH35j_QlrQhDWUN6A2Gg8iFayJ69xDEdHAVCGRzN3woEI2ozDR
s
.
-nBoKLH0YkLZPSI9
.
o4k2cnGN8rSSw3IDo1YuySkqeS_t2m1GXklSgqBdpACm6UJuJowOHC5ytjqYgR
L-I-soPlwqMUf4UgRWWeaOGNw6vGW-xyM01lTYxrXfVzIIaRdhYtEMRBvBWbEw
P7ua1DRfvaOjgZv6Ifa3brcAM64d8p5lhhNcizPersuhw5f-pGYzseva-TUaL8
iWnctc-sSwy7SQmRkfhDjwbz0fz6kFovEgj64X1I5s7E6GLp5fnbYGLa1QUiML
7Cc2GxgvI7zqWo0YIEc7aCflLG1-8BboVWFdZKLK9vNoycrYHumwzKluLWEbSV
maPpOslY2n525DxDfWaVFUfKQxMF56vn4B9QMpWAbnypNimbM8zVOw
.
UCGiqJxhBI3IFVdPalHHvA
```

Figure 92: Compact Serialization

The resulting JWE object using the JSON General Serialization:

```
{
  "recipients": [
    {
      "encrypted_key": "rT99rwrBTbTI7IJM8fU3Eli7226HEB7IchCxNu
          h7lCiud48LxeolRdtFF4nzQibeYOl5S_PJsAXZwSXtDePz9hk-Bb
          tsTBqC2UsPOdwjC9NhNupNNu9uHIVftDyucvI6hvALeZ6OGnhNV4
          v1zx2k7O1D89mAzfw-_kT3tkuorpDU-CpBENfIHX1Q58-Aad3FzM
          uo3Fn9buEP2yXakLXYa15BUXQsupM4A1GD4_H4Bd7V3u9h8Gkg8B
          pxKdUV9ScfJQTcYm6eJEBz3aSwIaK4T3-dwWpuBOhROQXBosJzS1
          asnuHtVMt2pKIIfux5BC6huIvmY7kzV7W7aIUrpYm_3H4zYvyMeq
          5pGqFmW2k8zpO878TRlZx7pZfPYDSXZyS0CfKKkMozT_qiCwZTSz
          4duYnt8hS4Z9sGthXn9uDqd6wycMagnQfOTs_lycTWmY-aqWVDKh
          jYNRf03NiwRtb5BE-tOdFwCASQj3uuAgPGrO2AWBe38UjQb0lvXn
          1SpyvYZ3WFc7WOJYaTa7A8DRn6MC6T-xDmMuxC0G7S2rscw5lQQU
          06MvZTlFOt0UvfuKBa03cxA_nIBIhLMjY2kOTxQMmpDPTr6Cbo8a
          KaOnx6ASE5Jx9paBpnNmOOKH35j_QlrQhDWUN6A2Gg8iFayJ69xD
          EdHAVCGRzN3woEI2ozDRs"
    }
  ],
  "protected": "eyJhbGciOiJSU0EtT0FFUCIsImtpZCI6InNhbXdpc2UuZ2
      FtZ2VlQGhvYmJpdG9uLmV4YW1wbGUiLCJlbmMiOiJBMjU2R0NNIn0",
  "iv": "-nBoKLH0YkLZPSI9",
  "ciphertext": "o4k2cnGN8rSSw3IDo1YuySkqeS_t2m1GXklSgqBdpACm6
      UJuJowOHC5ytjqYgRL-I-soPlwqMUf4UgRWWeaOGNw6vGW-xyM01lTYx
      rXfVzIIaRdhYtEMRBvBWbEwP7ua1DRfvaOjgZv6Ifa3brcAM64d8p5lh
      hNcizPersuhw5f-pGYzseva-TUaL8iWnctc-sSwy7SQmRkfhDjwbz0fz
      6kFovEgj64X1I5s7E6GLp5fnbYGLa1QUiML7Cc2GxgvI7zqWo0YIEc7a
      CflLG1-8BboVWFdZKLK9vNoycrYHumwzKluLWEbSVmaPpOslY2n525Dx
      DfWaVFUfKQxMF56vn4B9QMpWAbnypNimbM8zVOw",
  "tag": "UCGiqJxhBI3IFVdPalHHvA"
}
```


Figure 93: JSON General Serialization

The resulting JWE object using the JSON Flattened Serialization:

```
{
  "protected": "eyJhbGciOiJSU0EtT0FFUCIsImtpZCI6InNhbXdpc2UuZ2
      FtZ2VlQGhvYmJpdG9uLmV4YW1wbGUiLCJlbmMiOiJBMjU2R0NNIn0",
  "encrypted_key": "rT99rwrBTbTI7IJM8fU3Eli7226HEB7IchCxNuh7lC
      iud48LxeolRdtFF4nzQibeYOl5S_PJsAXZwSXtDePz9hk-BbtsTBqC2U
      sPOdwjC9NhNupNNu9uHIVftDyucvI6hvALeZ6OGnhNV4v1zx2k7O1D89
      mAzfw-_kT3tkuorpDU-CpBENfIHX1Q58-Aad3FzMuo3Fn9buEP2yXakL
      XYa15BUXQsupM4A1GD4_H4Bd7V3u9h8Gkg8BpxKdUV9ScfJQTcYm6eJE
      Bz3aSwIaK4T3-dwWpuBOhROQXBosJzS1asnuHtVMt2pKIIfux5BC6huI
      vmY7kzV7W7aIUrpYm_3H4zYvyMeq5pGqFmW2k8zpO878TRlZx7pZfPYD
      SXZyS0CfKKkMozT_qiCwZTSz4duYnt8hS4Z9sGthXn9uDqd6wycMagnQ
      fOTs_lycTWmY-aqWVDKhjYNRf03NiwRtb5BE-tOdFwCASQj3uuAgPGrO
      2AWBe38UjQb0lvXn1SpyvYZ3WFc7WOJYaTa7A8DRn6MC6T-xDmMuxC0G
      7S2rscw5lQQU06MvZTlFOt0UvfuKBa03cxA_nIBIhLMjY2kOTxQMmpDP
      Tr6Cbo8aKaOnx6ASE5Jx9paBpnNmOOKH35j_QlrQhDWUN6A2Gg8iFayJ
      69xDEdHAVCGRzN3woEI2ozDRs",
  "iv": "-nBoKLH0YkLZPSI9",
  "ciphertext": "o4k2cnGN8rSSw3IDo1YuySkqeS_t2m1GXklSgqBdpACm6
      UJuJowOHC5ytjqYgRL-I-soPlwqMUf4UgRWWeaOGNw6vGW-xyM01lTYx
      rXfVzIIaRdhYtEMRBvBWbEwP7ua1DRfvaOjgZv6Ifa3brcAM64d8p5lh
      hNcizPersuhw5f-pGYzseva-TUaL8iWnctc-sSwy7SQmRkfhDjwbz0fz
      6kFovEgj64X1I5s7E6GLp5fnbYGLa1QUiML7Cc2GxgvI7zqWo0YIEc7a
      CflLG1-8BboVWFdZKLK9vNoycrYHumwzKluLWEbSVmaPpOslY2n525Dx
      DfWaVFUfKQxMF56vn4B9QMpWAbnypNimbM8zVOw",
  "tag": "UCGiqJxhBI3IFVdPalHHvA"
}
```


Figure 94: JSON Flattened Serialization

5.3. Key Wrap using PBES2-AES-KeyWrap with AES-CBC-HMAC-SHA2

The example illustrates encrypting content using the "PBES2-HS512+A256KW" (PBES2 Password-based Encryption using HMAC-SHA-512 and AES-256-KeyWrap) key encryption algorithm with the "A128CBC-HS256" (AES-128-CBC-HMAC-SHA-256) content encryption algorithm.

A common use of password-based encryption is the import/export of keys. Therefore this example uses a JWK Set for the plaintext content instead of the plaintext from Figure 72.

Note that if password-based encryption is used for multiple recipients, it is expected that each recipient use different values for the PBES2 parameters "p2s" and "p2c".

Note that whitespace is added for readability as described in Section 1.1.

5.3.1. Input Factors

The following are supplied before beginning the encryption process:

- Plaintext content; this example uses the plaintext from Figure 95 (*NOTE* all whitespace added for readability)

- Password; this example uses the password from Figure 96 - with the sequence "\xe2\x80\x93" replaced with (U+2013 EN DASH)

- "alg" parameter of "PBES2-HS512+A256KW"

- "enc" parameter of "A128CBC-HS256"

```
{
  "keys": [
    {
      "kty": "oct",
      "kid": "77c7e2b8-6e13-45cf-8672-617b5b45243a",
      "use": "enc",
      "alg": "A128GCM",
      "k": "XctOhJAkA-pD9Lh7ZgW_2A"
    },
    {
      "kty": "oct",
      "kid": "81b20965-8332-43d9-a468-82160ad91ac8",
      "use": "enc",
      "alg": "A128KW",
      "k": "GZy6sIZ6wl9NJOKB-jnmVQ"
    },
    {
      "kty": "oct",
      "kid": "18ec08e1-bfa9-4d95-b205-2b4dd1d4321d",
      "use": "enc",
      "alg": "A256GCMKW",
      "k": "qC57l_uxcm7Nm3K-ct4GFjx8tM1U8CZ0NLBvdQstiS8"
    }
  ]
}
```

Figure 95: Plaintext Content

```
entrap_o\xe2\x80\x93peter_long\xe2\x80\x93credit_tun
```

Figure 96: Password

5.3.2. Generated Factors

The following are generated before encrypting:

- AES symmetric key as the Content Encryption Key (CEK); this example uses the key from Figure 97.
- Initialization vector/nonce; this example uses the initialization vector/nonce from Figure 98.

```
uwsjJXaBK407Qaf0_zpcpmr1Cs0CC50hIUEyGNEt3m0
```

Figure 97: Content Encryption Key, base64url-encoded

```
VBiCzVHNoLiR3F4V82uoTQ
```

Figure 98: Initialization Vector, base64url-encoded

5.3.3. Encrypting the Key

The following are generated before encrypting the CEK:

- Salt; this example uses the salt from Figure 99.
- Iteration count; this example uses the iteration count 8192.

```
8Q1SzinasR3xchYz6ZZcHA
```

Figure 99: Salt, base64url-encoded

Performing the key encryption operation over the CEK (Figure 97)) with the following:

- Password (Figure 96);
- Salt (Figure 99), encoded as an octet string; and
- Iteration count (8192)

produces the following encrypted key:

```
d3qNhUWfqheyPp4H8sjOWsDYajoej4c5Je6rlUtFPWdgtURtmeDV1g
```

Figure 100: Encrypted Key, base64url-encoded

5.3.4. Encrypting the Content

The following are generated before encrypting the content:

- JWE Protected Header; this example uses the header from Figure 101, encoded using [RFC4648] base64url to produce Figure 102.

```
{
  "alg": "PBES2-HS512+A256KW",
  "p2s": "8Q1SzinasR3xchYz6ZZcHA",
  "p2c": 8192,
  "cty": "jwk-set+json",
  "enc": "A128CBC-HS256"
}
```

Figure 101: JWE Protected Header JSON

```
eyJhbGciOiJQQkVTMi1IUzUxMitBMjU2S1ciLCJwMnMiOiI4UTFTemluYXNSM3
hjaFl6NlpaY0hBIiwicDJjIjo4MTkyLCJjdHkiOiJqd2stc2V0K2pzb24iLCJl
bmMiOiJBMTI4Q0JDLUhTMjU2In0
```

Figure 102: JWE Protected Header, base64url-encoded

Performing the content encryption operation over the Plaintext (Figure 95) with the the following:

- CEK (Figure 97);
- Initialization vector/nonce (Figure 98); and
- JWE Protected Header (Figure 102) as authenticated data

produces the following:

- Ciphertext from Figure 103.
- Authentication tag from Figure 104.

```
23i-Tb1AV4n0WKVSSgcQrdg6GRqsUKxjruHXYsTHAJLZ2nsnGIX86vMXqIi6IR
sfywCRFzLxEcZBRnTvG3nhzPk0GDD7FMyXhUHpDjEYCNA_XOmzg8yZR9oyjo6l
TF6si4q9FZ2EhzgFQCLO_6h5EVg3vR75_hkBsnuoqoM3dwejXBtIodN84PeqMb
6asmas_dpSsz7H10fC5ni9xIz424givB1YLldF6exVmL93R3fOoOJbmk2GBQZL
_SEGllv2cQsBgeprARsaQ7Bq99tT80coH8ItBjgV08AtzXFFsx9qKvC982KLKd
PQMTlVJKkqtV4Ru5LEVpBZXBnZrtViSOgyg6AiuwaS-rCrcD_ePOGSuxvgtrok
AKYPqmXUeRdjFJwafkYEkiuDCV9vWGAi1DH2xTafhJwcmywIyzi4BqRpmdn_N-
zl5tuJYyuvKhjKv6ihbsV_k1hJGPGAxJ6wUpmwC4PTQ2izEm0TuSE8oMKdTw8V
3kobXZ77ulMwDs4p
```

Figure 103: Ciphertext, base64url-encoded

```
0HlwodAhOCILG5SQ2LQ9dg
```

Figure 104: Authentication Tag, base64url-encoded

5.3.5. Output Results

The following compose the resulting JWE object:

- JWE Protected Header (Figure 102)
- Encrypted key (Figure 100)
- Initialization vector/nonce (Figure 98)
- Ciphertext (Figure 103)
- Authentication tag (Figure 104)

The resulting JWE object using the Compact serialization:

```
eyJhbGciOiJQQkVTMi1IUzUxMitBMjU2S1ciLCJwMnMiOiI4UTFTemluYXNSM3
hjaFl6NlpaY0hBIiwicDJjIjo4MTkyLCJjdHkiOiJqd2stc2V0K2pzb24iLCJl
bmMiOiJBMTI4Q0JDLUhTMjU2In0
.
d3qNhUWfqheyPp4H8sjOWsDYajoej4c5Je6rlUtFPWdgtURtmeDV1g
.
VBiCzVHNoLiR3F4V82uoTQ
.
23i-Tb1AV4n0WKVSSgcQrdg6GRqsUKxjruHXYsTHAJLZ2nsnGIX86vMXqIi6IR
sfywCRFzLxEcZBRnTvG3nhzPk0GDD7FMyXhUHpDjEYCNA_XOmzg8yZR9oyjo6l
TF6si4q9FZ2EhzgFQCLO_6h5EVg3vR75_hkBsnuoqoM3dwejXBtIodN84PeqMb
6asmas_dpSsz7H10fC5ni9xIz424givB1YLldF6exVmL93R3fOoOJbmk2GBQZL
_SEGllv2cQsBgeprARsaQ7Bq99tT80coH8ItBjgV08AtzXFFsx9qKvC982KLKd
PQMTlVJKkqtV4Ru5LEVpBZXBnZrtViSOgyg6AiuwaS-rCrcD_ePOGSuxvgtrok
AKYPqmXUeRdjFJwafkYEkiuDCV9vWGAi1DH2xTafhJwcmywIyzi4BqRpmdn_N-
zl5tuJYyuvKhjKv6ihbsV_k1hJGPGAxJ6wUpmwC4PTQ2izEm0TuSE8oMKdTw8V
3kobXZ77ulMwDs4p
.
0HlwodAhOCILG5SQ2LQ9dg
```

Figure 105: Compact Serialization

The resulting JWE object using the JSON General Serialization:

```
{
  "recipients": [
    {
      "encrypted_key": "d3qNhUWfqheyPp4H8sjOWsDYajoej4c5Je6rlU
          tFPWdgtURtmeDV1g"
    }
  ],
  "protected": "eyJhbGciOiJQQkVTMi1IUzUxMitBMjU2S1ciLCJwMnMiOi
      I4UTFTemluYXNSM3hjaFl6NlpaY0hBIiwicDJjIjo4MTkyLCJjdHkiOi
      Jqd2stc2V0K2pzb24iLCJlbmMiOiJBMTI4Q0JDLUhTMjU2In0",
  "iv": "VBiCzVHNoLiR3F4V82uoTQ",
  "ciphertext": "23i-Tb1AV4n0WKVSSgcQrdg6GRqsUKxjruHXYsTHAJLZ2
      nsnGIX86vMXqIi6IRsfywCRFzLxEcZBRnTvG3nhzPk0GDD7FMyXhUHpD
      jEYCNA_XOmzg8yZR9oyjo6lTF6si4q9FZ2EhzgFQCLO_6h5EVg3vR75_
      hkBsnuoqoM3dwejXBtIodN84PeqMb6asmas_dpSsz7H10fC5ni9xIz42
      4givB1YLldF6exVmL93R3fOoOJbmk2GBQZL_SEGllv2cQsBgeprARsaQ
      7Bq99tT80coH8ItBjgV08AtzXFFsx9qKvC982KLKdPQMTlVJKkqtV4Ru
      5LEVpBZXBnZrtViSOgyg6AiuwaS-rCrcD_ePOGSuxvgtrokAKYPqmXUe
      RdjFJwafkYEkiuDCV9vWGAi1DH2xTafhJwcmywIyzi4BqRpmdn_N-zl5
      tuJYyuvKhjKv6ihbsV_k1hJGPGAxJ6wUpmwC4PTQ2izEm0TuSE8oMKdT
      w8V3kobXZ77ulMwDs4p",
  "tag": "0HlwodAhOCILG5SQ2LQ9dg"
}
```


Figure 106: JSON General Serialization

The resulting JWE object using the JSON Flattened Serialization:

```
{
  "protected": "eyJhbGciOiJQQkVTMi1IUzUxMitBMjU2S1ciLCJwMnMiOi
      I4UTFTemluYXNSM3hjaFl6NlpaY0hBIiwicDJjIjo4MTkyLCJjdHkiOi
      Jqd2stc2V0K2pzb24iLCJlbmMiOiJBMTI4Q0JDLUhTMjU2In0",
  "encrypted_key": "d3qNhUWfqheyPp4H8sjOWsDYajoej4c5Je6rlUtFPW
      dgtURtmeDV1g",
  "iv": "VBiCzVHNoLiR3F4V82uoTQ",
  "ciphertext": "23i-Tb1AV4n0WKVSSgcQrdg6GRqsUKxjruHXYsTHAJLZ2
      nsnGIX86vMXqIi6IRsfywCRFzLxEcZBRnTvG3nhzPk0GDD7FMyXhUHpD
      jEYCNA_XOmzg8yZR9oyjo6lTF6si4q9FZ2EhzgFQCLO_6h5EVg3vR75_
      hkBsnuoqoM3dwejXBtIodN84PeqMb6asmas_dpSsz7H10fC5ni9xIz42
      4givB1YLldF6exVmL93R3fOoOJbmk2GBQZL_SEGllv2cQsBgeprARsaQ
      7Bq99tT80coH8ItBjgV08AtzXFFsx9qKvC982KLKdPQMTlVJKkqtV4Ru
      5LEVpBZXBnZrtViSOgyg6AiuwaS-rCrcD_ePOGSuxvgtrokAKYPqmXUe
      RdjFJwafkYEkiuDCV9vWGAi1DH2xTafhJwcmywIyzi4BqRpmdn_N-zl5
      tuJYyuvKhjKv6ihbsV_k1hJGPGAxJ6wUpmwC4PTQ2izEm0TuSE8oMKdT
      w8V3kobXZ77ulMwDs4p",
  "tag": "0HlwodAhOCILG5SQ2LQ9dg"
}
```


Figure 107: JSON Flattened Serialization

5.4. Key Agreement with Key Wrapping using ECDH-ES and AES-KeyWrap with AES-GCM

This example illustrates encrypting content using the "ECDH-ES+A128KW" (Elliptic Curve Diffie-Hellman Ephemeral-Static with AES-128-KeyWrap) key encryption algorithm and the "A128GCM" (AES-GCM) content encryption algorithm.

Note that only the EC public key is necessary to perform the key agreement. However, the example includes the EC private key to allow readers to validate the output.

Note that whitespace is added for readability as described in Section 1.1.

5.4.1. Input Factors

The following are supplied before beginning the encryption process:

- Plaintext content; this example uses the content from Figure 72
- EC public key; this example uses the public key from Figure 108
- "alg" parameter of "ECDH-ES+A128KW"
- "enc" parameter of "A128GCM"

```
{
  "kty": "EC",
  "kid": "peregrin.took@tuckborough.example",
  "use": "enc",
  "crv": "P-384",
  "x": "YU4rRUzdmVqmRtWOs2OpDE_T5fsNIodcG8G5FWPrTPMyxpzsSOGaQL
      pe2FpxBmu2",
  "y": "A8-yxCHxkfBz3hKZfI1jUYMjUhsEveZ9THuwFjH2sCNdtksRJU7D5-
      SkgaFL1ETP",
  "d": "iTx2pk7wW-GqJkHcEkFQb2EFyYcO7RugmaW3mRrQVAOUiPommT0Idn
      YK2xDlZh-j"
}
```


Figure 108: Elliptic Curve P-384 Key, in JWK format

(*NOTE*: While the key includes the private parameters, only the public parameters "crv", "x", and "y" are necessary for the encryption operation.)

5.4.2. Generated Factors

The following are generated before encrypting:

- Symmetric AES key as the Content Encryption Key (CEK); this example uses the key from Figure 109.
- Initialization vector/nonce; this example uses the initialization vector/nonce from Figure 110


```
Nou2ueKlP70ZXDbq9UrRwg
```


Figure 109: Content Encryption Key, base64url-encoded


```
mH-G2zVqgztUtnW_
```


Figure 110: Initialization Vector, base64url-encoded

5.4.3. Encrypting the Key

To encrypt the Content Encryption Key, the following are generated:

- Ephemeral EC private key on the same curve as the EC public key; this example uses the private key from Figure 111.

```
{
  "kty": "EC",
  "crv": "P-384",
  "x": "uBo4kHPw6kbjx5l0xowrd_oYzBmaz-GKFZu4xAFFkbYiWgutEK6iuE
      DsQ6wNdNg3",
  "y": "sp3p5SGhZVC2faXumI-e9JU2Mo8KpoYrFDr5yPNVtW4PgEwZOyQTA-
      JdaY8tb7E0",
  "d": "D5H4Y_5PSKZvhfVFbcCYJOtcGZygRgfZkpsBr59Icmmhe9sW6nkZ8W
      fwhinUfWJg"
}
```

Figure 111: Ephemeral Elliptic Curve P-384 Key, in JWK format

Performing the key encryption operation over the CEK (Figure 109) with the following:

- The static Elliptic Curve public key (Figure 108); and
- The ephemeral Elliptic Curve private key (Figure 111);

produces the following JWE encrypted key:

```
0DJjBXri_kBcC46IkU5_Jk9BqaQeHdv2
```

Figure 112: Encrypted Key, base64url-encoded

5.4.4. Encrypting the Content

The following are generated before encrypting the content:

- JWE Protected Header; this example uses the header from Figure 113, encoded to [RFC4648] base64url as Figure 114.

```
{
  "alg": "ECDH-ES+A128KW",
  "kid": "peregrin.took@tuckborough.example",
  "epk": {
    "kty": "EC",
    "crv": "P-384",
    "x": "uBo4kHPw6kbjx5l0xowrd_oYzBmaz-GKFZu4xAFFkbYiWgutEK6i
        uEDsQ6wNdNg3",
    "y": "sp3p5SGhZVC2faXumI-e9JU2Mo8KpoYrFDr5yPNVtW4PgEwZOyQT
        A-JdaY8tb7E0"
  },
  "enc": "A128GCM"
}
```

Figure 113: JWE Protected Header JSON

```
eyJhbGciOiJFQ0RILUVTK0ExMjhLVyIsImtpZCI6InBlcmVncmluLnRvb2tAdH
Vja2Jvcm91Z2guZXhhbXBsZSIsImVwayI6eyJrdHkiOiJFQyIsImNydiI6IlAt
Mzg0IiwieCI6InVCbzRrSFB3Nmtiang1bDB4b3dyZF9vWXpCbWF6LUdLRlp1NH
hBRkZrYllpV2d1dEVLNml1RURzUTZ3TmROZzMiLCJ5Ijoic3AzcDVTR2haVkMy
ZmFYdW1JLWU5SlUyTW84S3BvWXJGRHI1eVBOVnRXNFBnRXdaT3lRVEEtSmRhWT
h0YjdFMCJ9LCJlbmMiOiJBMTI4R0NNIn0
```

Figure 114: JWE Protected Header, base64url-encoded

Performing the content encryption operation on the Plaintext (Figure 72) using the following:

- CEK (Figure 109);
- Initialization vector/nonce (Figure 110); and
- JWE Protected Header (Figure 114) as authenticated data

produces the following:

- Ciphertext from Figure 115.
- Authentication tag from Figure 116.

```
tkZuOO9h95OgHJmkkrfLBisku8rGf6nzVxhRM3sVOhXgz5NJ76oID7lpnAi_cP
WJRCjSpAaUZ5dOR3Spy7QuEkmKx8-3RCMhSYMzsXaEwDdXta9Mn5B7cCBoJKB0
IgEnj_qfo1hIi-uEkUpOZ8aLTZGHfpl05jMwbKkTe2yK3mjF6SBAsgicQDVCkc
Y9BLluzx1RmC3ORXaM0JaHPB93YcdSDGgpgBWMVrNU1ErkjcMqMoT_wtCex3w0
3XdLkjXIuEr2hWgeP-nkUZTPU9EoGSPj6fAS-bSz87RCPrxZdj_iVyC6QWcqAu
07WNhjzJEPc4jVntRJ6K53NgPQ5p99l3Z408OUqj4ioYezbS6vTPlQ
```

Figure 115: Ciphertext, base64url-encoded

```
WuGzxmcreYjpHGJoa17EBg
```

Figure 116: Authentication Tag, base64url-encoded

5.4.5. Output Results

The following compose the resulting JWE object:

- JWE Protected Header (Figure 114)
- Encrypted key (Figure 112)
- Initialization vector/nonce (Figure 110)
- Ciphertext (Figure 115)

o Authentication tag (Figure 116)

The resulting JWE object using the Compact serialization:

```
eyJhbGciOiJFQ0RILUVTK0ExMjhLVyIsImtpZCI6InBlcmVncmluLnRvb2tAdH
Vja2Jvcm91Z2guZXhhbXBsZSIsImVwayI6eyJrdHkiOiJFQyIsImNydiI6IlAt
Mzg0IiwieCI6InVCbzRrSFB3Nmtiang1bDB4b3dyZF9vWXpCbWF6LUdLRlp1NH
hBRkZrYllpV2d1dEVLNml1RURzUTZ3TmROZzMiLCJ5Ijoic3AzcDVTR2haVkMy
ZmFYdW1JLWU5SlUyTW84S3BvWXJGRHI1eVBOVnRXNFBnRXdaT3lRVEEtSmRhWT
h0YjdFMCJ9LCJlbmMiOiJBMTI4R0NNIn0
.
0DJjBXri_kBcC46IkU5_Jk9BqaQeHdv2
.
mH-G2zVqgztUtnW_
.
tkZuOO9h95OgHJmkkrfLBisku8rGf6nzVxhRM3sVOhXgz5NJ76oID7lpnAi_cP
WJRCjSpAaUZ5dOR3Spy7QuEkmKx8-3RCMhSYMzsXaEwDdXta9Mn5B7cCBoJKB0
IgEnj_qfo1hIi-uEkUpOZ8aLTZGHfpl05jMwbKkTe2yK3mjF6SBAsgicQDVCkc
Y9BLluzx1RmC3ORXaM0JaHPB93YcdSDGgpgBWMVrNU1ErkjcMqMoT_wtCex3w0
3XdLkjXIuEr2hWgeP-nkUZTPU9EoGSPj6fAS-bSz87RCPrxZdj_iVyC6QWcqAu
07WNhjzJEPc4jVntRJ6K53NgPQ5p99l3Z408OUqj4ioYezbS6vTPlQ
.
WuGzxmcreYjpHGJoa17EBg
```

Figure 117: Compact Serialization

The resulting JWE object using the JSON General Serialization:

```
{
  "recipients": [
    {
      "encrypted_key": "0DJjBXri_kBcC46IkU5_Jk9BqaQeHdv2"
    }
  ],
  "protected": "eyJhbGciOiJFQ0RILUVTK0ExMjhLVyIsImtpZCI6InBlcm
      VncmluLnRvb2tAdHVja2Jvcm91Z2guZXhhbXBsZSIsImVwayI6eyJrdH
      kiOiJFQyIsImNydiI6IlAtMzg0IiwieCI6InVCbzRrSFB3Nmtiang1bD
      B4b3dyZF9vWXpCbWF6LUdLRlp1NHhBRkZrYllpV2d1dEVLNml1RURzUT
      Z3TmROZzMiLCJ5Ijoic3AzcDVTR2haVkMyZmFYdW1JLWU5SlUyTW84S3
      BvWXJGRHI1eVBOVnRXNFBnRXdaT3lRVEEtSmRhWTh0YjdFMCJ9LCJlbm
      MiOiJBMTI4R0NNIn0",
  "iv": "mH-G2zVqgztUtnW_",
  "ciphertext": "tkZuOO9h95OgHJmkkrfLBisku8rGf6nzVxhRM3sVOhXgz
      5NJ76oID7lpnAi_cPWJRCjSpAaUZ5dOR3Spy7QuEkmKx8-3RCMhSYMzs
      XaEwDdXta9Mn5B7cCBoJKB0IgEnj_qfo1hIi-uEkUpOZ8aLTZGHfpl05
      jMwbKkTe2yK3mjF6SBAsgicQDVCkcY9BLluzx1RmC3ORXaM0JaHPB93Y
      cdSDGgpgBWMVrNU1ErkjcMqMoT_wtCex3w03XdLkjXIuEr2hWgeP-nkU
      ZTPU9EoGSPj6fAS-bSz87RCPrxZdj_iVyC6QWcqAu07WNhjzJEPc4jVn
      tRJ6K53NgPQ5p99l3Z408OUqj4ioYezbS6vTPlQ",
  "tag": "WuGzxmcreYjpHGJoa17EBg"
}
```


Figure 118: JSON General Serialization

The resulting JWE object using the JSON Flattened Serialization:

```
{
  "protected": "eyJhbGciOiJFQ0RILUVTK0ExMjhLVyIsImtpZCI6InBlcm
      VncmluLnRvb2tAdHVja2Jvcm91Z2guZXhhbXBsZSIsImVwayI6eyJrdH
      kiOiJFQyIsImNydiI6IlAtMzg0IiwieCI6InVCbzRrSFB3Nmtiang1bD
      B4b3dyZF9vWXpCbWF6LUdLRlp1NHhBRkZrYllpV2d1dEVLNml1RURzUT
      Z3TmROZzMiLCJ5Ijoic3AzcDVTR2haVkMyZmFYdW1JLWU5SlUyTW84S3
      BvWXJGRHI1eVBOVnRXNFBnRXdaT3lRVEEtSmRhWTh0YjdFMCJ9LCJlbm
      MiOiJBMTI4R0NNIn0",
  "encrypted_key": "0DJjBXri_kBcC46IkU5_Jk9BqaQeHdv2",
  "iv": "mH-G2zVqgztUtnW_",
  "ciphertext": "tkZuOO9h95OgHJmkkrfLBisku8rGf6nzVxhRM3sVOhXgz
      5NJ76oID7lpnAi_cPWJRCjSpAaUZ5dOR3Spy7QuEkmKx8-3RCMhSYMzs
      XaEwDdXta9Mn5B7cCBoJKB0IgEnj_qfo1hIi-uEkUpOZ8aLTZGHfpl05
      jMwbKkTe2yK3mjF6SBAsgicQDVCkcY9BLluzx1RmC3ORXaM0JaHPB93Y
      cdSDGgpgBWMVrNU1ErkjcMqMoT_wtCex3w03XdLkjXIuEr2hWgeP-nkU
      ZTPU9EoGSPj6fAS-bSz87RCPrxZdj_iVyC6QWcqAu07WNhjzJEPc4jVn
      tRJ6K53NgPQ5p99l3Z408OUqj4ioYezbS6vTPlQ",
  "tag": "WuGzxmcreYjpHGJoa17EBg"
}
```

Figure 119: JSON Flattened Serialization

5.5. Key Agreement using ECDH-ES with AES-CBC-HMAC-SHA2

This example illustrates encrypting content using the "ECDH-ES" (Elliptic Curve Diffie-Hellman Ephemeral-Static) key agreement algorithm and the "A128CBC-HS256" (AES-128-CBC-HMAC-SHA-256) content encryption algorithm.

Note that only the EC public key is necessary to perform the key agreement. However, the example includes the EC private key to allow readers to validate the output.

Note that whitespace is added for readability as described in Section 1.1.

5.5.1. Input Factors

The following are supplied before beginning the encryption process:

- Plaintext content; this example uses the content from Figure 72.
- EC public key; this example uses the public key from Figure 120.
- "alg" parameter of "ECDH-ES"
- "enc" parameter of "A128CBC-HS256"

```
{
  "kty": "EC",
  "kid": "meriadoc.brandybuck@buckland.example",
  "use": "enc",
  "crv": "P-256",
  "x": "Ze2loSV3wrroKUN_4zhwGhCqo3Xhu1td4QjeQ5wIVR0",
  "y": "HlLtdXARY_f55A3fnzQbPcm6hgr34Mp8p-nuzQCE0Zw",
  "d": "r_kHyZ-a06rmxM3yESK84r1otSg-aQcVStkRhA-iCM8"
}
```

Figure 120: Elliptic Curve P-256 Key

(*NOTE*: While the key includes the private parameters, only the public parameters "crv", "x", and "y" are necessary for the encryption operation.)

5.5.2. Generated Factors

The following are generated before encrypting:

- Initialization vector/nonce; this examples uses the initialization vector/nonce from Figure 121.

```
yc9N8v5sYyv3iGQT926IUg
```

Figure 121: Initialization Vector, base64url-encoded

NOTE: The Content Encryption Key (CEK) is not randomly generated; instead it is determined using ECDH-ES key agreement.

5.5.3. Key Agreement

The following are generated to agree on a CEK:

- Ephemeral private key; this example uses the private key from Figure 122.

```
{
  "kty": "EC",
  "crv": "P-256",
  "x": "mPUKT_bAWGHIhg0TpjjqVsP1rXWQu_vwVOHHtNkdYoA",
  "y": "8BQAsImGeAS46fyWw5MhYfGTT0IjBpFw2SS34Dv4Irs",
  "d": "AtH35vJsQ9SGjYfOsjUxYXQKrPH3FjZHmEtSKoSN8cM"
}
```

Figure 122: Ephemeral public key, in JWK format

Performing the ECDH operation using the static EC public key (Figure 120) over the ephemeral private key Figure 122) produces the following CEK:

```
hzHdlfQIAEehb8Hrd_mFRhKsKLEzPfshfXs9l6areCc
```

Figure 123: Agreed to Content Encryption Key, base64url-encoded

5.5.4. Encrypting the Content

The following are generated before encrypting the content:

- JWE Protected Header; this example uses the header from Figure 124, encoded to [RFC4648] as Figure 125.

```
{
  "alg": "ECDH-ES",
  "kid": "meriadoc.brandybuck@buckland.example",
  "epk": {
    "kty": "EC",
    "crv": "P-256",
    "x": "mPUKT_bAWGHIhg0TpjjqVsP1rXWQu_vwVOHHtNkdYoA",
    "y": "8BQAsImGeAS46fyWw5MhYfGTT0IjBpFw2SS34Dv4Irs"
  },
  "enc": "A128CBC-HS256"
}
```

Figure 124: JWE Protected Header JSON

```
eyJhbGciOiJFQ0RILUVTIiwia2lkIjoibWVyaWFkb2MuYnJhbmR5YnVja0BidW
NrbGFuZC5leGFtcGxlIiwiZXBrIjp7Imt0eSI6IkVDIiwiY3J2IjoiUC0yNTYi
LCJ4IjoibVBVS1RfYkFXR0hJaGcwVHBqanFWc1AxclhXUXVfdndWT0hIdE5rZF
lvQSIsInkiOiI4QlFBc0ltR2VBUzQ2ZnlXdzVNaFlmR1RUMElqQnBGdzJTUzM0
RHY0SXJzIn0sImVuYyI6IkExMjhDQkMtSFMyNTYifQ
```

Figure 125: JWE Protected Header, base64url-encoded

Performing the content encryption operation on the Plaintext (Figure 72) using the following:

- CEK (Figure 123);
- Initialization vector/nonce (Figure 121); and
- JWE Protected Header (Figure 125) as authenticated data

produces the following:

- Ciphertext from Figure 126.
- Authentication tag from Figure 127.

```
BoDlwPnTypYq-ivjmQvAYJLb5Q6l-F3LIgQomlz87yW4OPKbWE1zSTEFjDfhU9
IPIOSA9Bml4m7iDFwA-1ZXvHteLDtw4R1XRGMEsDIqAYtskTTmzmzNa-_q4F_e
vAPUmwlO-ZG45Mnq4uhM1fm_D9rBtWolqZSF3xGNNkpOMQKF1Cl8i8wjzRli7-
IXgyirlKQsbhhqRzkv8IcY6aHl24j03C-AR2le1r7URUhArM79BY8soZU0lzwI
-sD5PZ3l4NDCCei9XkoIAfsXJWmySPoeRb2Ni5UZL4mYpvKDiwmyzGd65KqVw7
MsFfI_K767G9C9Azp73gKZD0DyUn1mn0WW5LmyX_yJ-3AROq8p1WZBfG-ZyJ61
95_JGG2m9Csg
```

Figure 126: Ciphertext, base64url-encoded

```
WCCkNa-x4BeB9hIDIfFuhg
```

Figure 127: Authentication Tag, base64url-encoded

5.5.5. Output Results

The following compose the resulting JWE object:

- JWE Protected Header (Figure 114)
- Initialization vector/nonce (Figure 110)
- Ciphertext (Figure 115)
- Authentication tag (Figure 116)

Only the JSON General Serialization is presented because the JSON Flattened Serialization is identical.

the resulting JWE object using the Compact serialization:

```
eyJhbGciOiJFQ0RILUVTIiwia2lkIjoibWVyaWFkb2MuYnJhbmR5YnVja0BidW
NrbGFuZC5leGFtcGxlIiwiZXBrIjp7Imt0eSI6IkVDIiwiY3J2IjoiUC0yNTYi
LCJ4IjoibVBVS1RfYkFXR0hJaGcwVHBqanFWc1AxclhXUXVfdndWT0hIdE5rZF
lvQSIsInkiOiI4QlFBc0ltR2VBUzQ2ZnlXdzVNaFlmR1RUMElqQnBGdzJTUzM0
RHY0SXJzIn0sImVuYyI6IkExMjhDQkMtSFMyNTYifQ
.
.
yc9N8v5sYyv3iGQT926IUg
.
BoDlwPnTypYq-ivjmQvAYJLb5Q6l-F3LIgQomlz87yW4OPKbWE1zSTEFjDfhU9
IPIOSA9Bml4m7iDFwA-1ZXvHteLDtw4R1XRGMEsDIqAYtskTTmzmzNa-_q4F_e
vAPUmwlO-ZG45Mnq4uhM1fm_D9rBtWolqZSF3xGNNkpOMQKF1Cl8i8wjzRli7-
IXgyirlKQsbhhqRzkv8IcY6aHl24j03C-AR2le1r7URUhArM79BY8soZU0lzwI
-sD5PZ3l4NDCCei9XkoIAfsXJWmySPoeRb2Ni5UZL4mYpvKDiwmyzGd65KqVw7
MsFfI_K767G9C9Azp73gKZD0DyUn1mn0WW5LmyX_yJ-3AROq8p1WZBfG-ZyJ61
95_JGG2m9Csg
.
WCCkNa-x4BeB9hIDIfFuhg
```

Figure 128: Compact Serialization

the resulting JWE object using the JSON General Serialization:

```
{
  "protected": "eyJhbGciOiJFQ0RILUVTIiwia2lkIjoibWVyaWFkb2MuYn
      JhbmR5YnVja0BidWNrbGFuZC5leGFtcGxlIiwiZXBrIjp7Imt0eSI6Ik
      VDIiwiY3J2IjoiUC0yNTYiLCJ4IjoibVBVS1RfYkFXR0hJaGcwVHBqan
      FWc1AxclhXUXVfdndWT0hIdE5rZFlvQSIsInkiOiI4QlFBc0ltR2VBUz
      Q2ZnlXdzVNaFlmR1RUMElqQnBGdzJTUzM0RHY0SXJzIn0sImVuYyI6Ik
      ExMjhDQkMtSFMyNTYifQ",
  "iv": "yc9N8v5sYyv3iGQT926IUg",
  "ciphertext": "BoDlwPnTypYq-ivjmQvAYJLb5Q6l-F3LIgQomlz87yW4O
      PKbWE1zSTEFjDfhU9IPIOSA9Bml4m7iDFwA-1ZXvHteLDtw4R1XRGMEs
      DIqAYtskTTmzmzNa-_q4F_evAPUmwlO-ZG45Mnq4uhM1fm_D9rBtWolq
      ZSF3xGNNkpOMQKF1Cl8i8wjzRli7-IXgyirlKQsbhhqRzkv8IcY6aHl2
      4j03C-AR2le1r7URUhArM79BY8soZU0lzwI-sD5PZ3l4NDCCei9XkoIA
      fsXJWmySPoeRb2Ni5UZL4mYpvKDiwmyzGd65KqVw7MsFfI_K767G9C9A
      zp73gKZD0DyUn1mn0WW5LmyX_yJ-3AROq8p1WZBfG-ZyJ6195_JGG2m9
      Csg",
  "tag": "WCCkNa-x4BeB9hIDIfFuhg"
}
```

Figure 129: JSON General Serialization

5.6. Direct Encryption using AES-GCM

This example illustrates encrypting content using a previously exchanged key directly and the "A128GCM" (AES-GCM) content encryption algorithm.

Note that whitespace is added for readability as described in Section 1.1.

5.6.1. Input Factors

The following are supplied before beginning the encryption process:

- Plaintext content; this example uses the content from Figure 72.
- AES symmetric key as the Content Encryption Key (CEK); this example uses the key from Figure 130.
- "alg" parameter of "dir"
- "enc" parameter of "A128GCM"

```
{
  "kty": "oct",
  "kid": "77c7e2b8-6e13-45cf-8672-617b5b45243a",
  "use": "enc",
  "alg": "A128GCM",
  "k": "XctOhJAkA-pD9Lh7ZgW_2A"
}
```

Figure 130: AES 128-bit key, in JWK format

5.6.2. Generated Factors

The following are generated before encrypting:

- Initialization vector/nonce; this example uses the initialization vector/nonce from Figure 131.

```
refa467QzzKx6QAB
```

Figure 131: Initialization Vector, base64url-encoded

5.6.3. Encrypting the Content

The following are generated before encrypting the content:

o JWE Protected Header; this example uses the header from Figure 132, encoded as [RFC4648] base64url to produce Figure 133.

```
{
  "alg": "dir",
  "kid": "77c7e2b8-6e13-45cf-8672-617b5b45243a",
  "enc": "A128GCM"
}
```

Figure 132: JWE Protected Header JSON

Encoded as [RFC4648] base64url:

```
eyJhbGciOiJkaXIiLCJraWQiOiI3N2M3ZTJiOC02ZTEzLTQ1Y2YtODY3Mi02MT
diNWI0NTI0M2EiLCJlbmMiOiJBMTI4R0NNIn0
```

Figure 133: JWE Protected Header, base64url-encoded

Performing the encryption operation on the Plaintext (Figure 72) using the following:

o CEK (Figure 130);

o Initialization vector/nonce (Figure 131); and

o JWE Protected Header (Figure 133) as authenticated data

produces the following:

o Ciphertext from Figure 134.

o Authentication tag from Figure 135.

```
JW_i_f52hww_ELQPGaYyeAB6HYGcR559l9TYnSovc23XJoBcW29rHP8yZOZG7Y
hLpT1bjFuvZPjQS-m0IFtVcXkZXdH_lr_FrdYt9HRUYkshtrMmIUAyGmUnd9zM
DB2n0cRDIHAzFVeJUDxkUwVAE7_YGRPdcqMyiBoCO-FBdE-Nceb4h3-FtBP-c_
BIwCPTjb9o0SbdcdREEMJMyZBH8ySWMVi1gPD9yxi-aQpGbSv_F9N4IZAxscj5
g-NJsUPbjk29-s7LJAGb15wEBtXphVCgyy53CoIKLHHeJHXex45Uz9aKZSRSIn
ZI-wjsY0yu3cT4_aQ3i1o-tiE-F8Ios61EKgyIQ4CWao8PFMj8TTnp
```

Figure 134: Ciphertext, base64url-encoded

```
vbb32Xvllea2OtmHAdccRQ
```

Figure 135: Authentication Tag, base64url-encoded

5.6.4. Output Results

The following compose the resulting JWE object:

- JWE Protected Header (Figure 133)
- Initialization vector/nonce (Figure 131)
- Ciphertext (Figure 134)
- Authentication tag (Figure 135)

Only the JSON General Serialization is presented because the JSON Flattened Serialization is identical.

The resulting JWE object using the Compact serialization:

```
eyJhbGciOiJkaXIiLCJraWQiOiI3N2M3ZTJiOC02ZTEzLTQ1Y2YtODY3Mi02MT
diNWI0NTI0M2EiLCJlbmMiOiJBMTI4R0NNIn0
.
.
refa467QzzKx6QAB
.
JW_i_f52hww_ELQPGaYyeAB6HYGcR559l9TYnSovc23XJoBcW29rHP8yZOZG7Y
hLpT1bjFuvZPjQS-m0IFtVcXkZXdH_lr_FrdYt9HRUYkshtrMmIUAyGmUnd9zM
DB2n0cRDIHAzFVeJUDxkUwVAE7_YGRPdcqMyiBoCO-FBdE-Nceb4h3-FtBP-c_
BIwCPTjb9o0SbdcdREEMJMyZBH8ySWMVi1gPD9yxi-aQpGbSv_F9N4IZAxscj5
g-NJsUPbjk29-s7LJAGb15wEBtXphVCgyy53CoIKLHHeJHXex45Uz9aKZSRSIn
ZI-wjsY0yu3cT4_aQ3i1o-tiE-F8Ios61EKgyIQ4CWao8PFMj8TTnp
.
vbb32Xvllea2OtmHAdccRQ
```

Figure 136: Compact Serialization

The resulting JWE object using the JSON General Serialization:

```
{
  "protected": "eyJhbGciOiJkaXIiLCJraWQiOiI3N2M3ZTJiOC02ZTEzLT
      Q1Y2YtODY3Mi02MTdiNWI0NTI0M2EiLCJlbmMiOiJBMTI4R0NNIn0",
  "iv": "refa467QzzKx6QAB",
  "ciphertext": "JW_i_f52hww_ELQPGaYyeAB6HYGcR559l9TYnSovc23XJ
      oBcW29rHP8yZOZG7YhLpT1bjFuvZPjQS-m0IFtVcXkZXdH_lr_FrdYt9
      HRUYkshtrMmIUAyGmUnd9zMDB2n0cRDIHAzFVeJUDxkUwVAE7_YGRPdc
      qMyiBoCO-FBdE-Nceb4h3-FtBP-c_BIwCPTjb9o0SbdcdREEMJMyZBH8
      ySWMVi1gPD9yxi-aQpGbSv_F9N4IZAxscj5g-NJsUPbjk29-s7LJAGb1
      5wEBtXphVCgyy53CoIKLHHeJHXex45Uz9aKZSRSInZI-wjsY0yu3cT4_
      aQ3i1o-tiE-F8Ios61EKgyIQ4CWao8PFMj8TTnp",
  "tag": "vbb32Xvllea2OtmHAdccRQ"
}
```


Figure 137: JSON General Serialization

5.7. Key Wrap using AES-GCM KeyWrap with AES-CBC-HMAC-SHA2

This example illustrates encrypting content using the "A256GCMKW" (AES-256-GCM-KeyWrap) key encryption algorithm with the "A128CBC-HS256" (AES-128-CBC-HMAC-SHA-256) content encryption algorithm.

Note that whitespace is added for readability as described in Section 1.1.

5.7.1. Input Factors

The following are supplied before beginning the encryption process:

- Plaintext content; this example uses the content from Figure 72.
- AES symmetric key; this example uses the key from Figure 138.
- "alg" parameter of "A256GCMKW"
- "enc" parameter of "A128CBC-HS256"


```
{
  "kty": "oct",
  "kid": "18ec08e1-bfa9-4d95-b205-2b4dd1d4321d",
  "use": "enc",
  "alg": "A256GCMKW",
  "k": "qC57l_uxcm7Nm3K-ct4GFjx8tM1U8CZ0NLBvdQstiS8"
}
```


Figure 138: AES 256-bit Key

5.7.2. Generated Factors

The following are generated before encrypting:

- AES symmetric key as the Content Encryption Key (CEK); this example uses the key from Figure 139.
- Initialization vector/nonce for content encryption; this example uses the initialization vector/nonce from Figure 140.

```
UWxARpat23nL9ReIj4WG3D1ee9I4r-Mv5QLuFXdy_rE
```

Figure 139: Content Encryption Key, base64url-encoded

```
gz6NjyEFNm_vm8Gj6FwoFQ
```

Figure 140: Initialization Vector, base64url-encoded

5.7.3. Encrypting the Key

The following are generated before encrypting the CEK:

- Initialization vector/nonce for key wrapping; this example uses the initialization vector/nonce from Figure 141.

```
KkYT0GX_2jHlfqN_
```

Figure 141: Key Wrap Initialization Vector, base64url-encoded

Performing the key encryption operation over the CEK (Figure 139) with the following:

- AES symmetric key (Figure 138);
- Key wrap initialization vector/nonce (Figure 141); and
- The empty string as authenticated data

produces the following:

- Encrypted Key from Figure 142.
- Key wrap authentication tag from Figure 143.

```
lJf3HbOApxMEBkCMOoTnnABxs_CvTWUmZQ2ElLvYNok
```

Figure 142: Encrypted Key, base64url-encoded

```
kfPduVQ3T3H6vnewt--ksw
```

Figure 143: Key Wrap Authentication Tag, base64url-encoded

5.7.4. Encrypting the Content

The following are generated before encrypting the content:

- JWE Protected Header; this example uses the header from Figure 144, encoded to [RFC4648] base64url as Figure 145.

```
{
  "alg": "A256GCMKW",
  "kid": "18ec08e1-bfa9-4d95-b205-2b4dd1d4321d",
  "tag": "kfPduVQ3T3H6vnewt--ksw",
  "iv": "KkYT0GX_2jHlfqN_",
  "enc": "A128CBC-HS256"
}
```

Figure 144: JWE Protected Header JSON

```
eyJhbGciOiJBMjU2R0NNS1ciLCJraWQiOiIxOGVjMDhlMS1iZmE5LTRkOTUtYj
IwNS0yYjRkZDFkNDMyMWQiLCJ0YWciOiJrZlBkdVZRM1QzSDZ2bmV3dC0ta3N3
IiwiaXYiOiJLa1lUMEdYXzJqSGxmcU5fIiwiZW5jIjoiQTEyOENCQy1IUzI1Ni
J9
```

Figure 145: JWE Protected Header, base64url-encoded

Performing the content encryption operation over the Plaintext (Figure 72) with the following:

- CEK (Figure 139);
- Initialization vector/nonce (Figure 140); and
- JWE Protected Header (Figure 145) as authenticated data

produces the following:

- Ciphertext from Figure 146.
- Authentication tag from Figure 147.

```
Jf5p9-ZhJlJy_IQ_byKFmI0Ro7w7G1QiaZpI8OaiVgD8EqoDZHyFKFBupS8iaE
eVIgMqWmsuJKuoVgzR3YfzoMd3GxEm3VxNhzWyWtZKX0gxKdy6HgLvqoGNbZCz
LjqcpDiF8q2_62EVAbr2uSc2oaxFmFuIQHLcqAHxy51449xkjZ7ewzZaGV3eFq
hpco8o4DijXaG5_7kp3h2cajRfDgymuxUbWgLqaeNQaJtvJmSMFuEOSAzw9Hde
b6yhdTynCRmu-kqtO5Dec4lT2OMZKpnxc_F1_4yDJFcqb5CiDSmA-psB2k0Jtj
xAj4UPI61oONK7zzFIu4gBfjJCndsZfdvG7h8wGjV98QhrKEnR7xKZ3KCr0_qR
1B-gxpNk3xWU
```

Figure 146: Ciphertext, base64url-encoded

```
DKW7jrb4WaRSNfbXVPlT5g
```

Figure 147: Authentication Tag, base64url-encoded

5.7.5. Output Results

The following compose the resulting JWE object:

- JWE Protected Header (Figure 145)
- encrypted key (Figure 142)
- Initialization vector/nonce (Figure 140)
- Ciphertext (Figure 146)
- Authentication tag (Figure 147)

The resulting JWE object using the Compact serialization:

```
eyJhbGciOiJBMjU2R0NNS1ciLCJraWQiOiIxOGVjMDhlMS1iZmE5LTRkOTUtYj
IwNS0yYjRkZDFkNDMyMWQiLCJ0YWciOiJrZlBkdVZRM1QzSDZ2bmV3dC0ta3N3
IiwiaXYiOiJLa1lUMEdYXzJqSGxmcU5fIiwiZW5jIjoiQTEyOENCQy1IUzI1Ni
J9
.
lJf3HbOApxMEBkCMOoTnnABxs_CvTWUmZQ2ElLvYNok
.
gz6NjyEFNm_vm8Gj6FwoFQ
.
Jf5p9-ZhJlJy_IQ_byKFmI0Ro7w7G1QiaZpI8OaiVgD8EqoDZHyFKFBupS8iaE
eVIgMqWmsuJKuoVgzR3YfzoMd3GxEm3VxNhzWyWtZKX0gxKdy6HgLvqoGNbZCz
LjqcpDiF8q2_62EVAbr2uSc2oaxFmFuIQHLcqAHxy51449xkjZ7ewzZaGV3eFq
hpco8o4DijXaG5_7kp3h2cajRfDgymuxUbWgLqaeNQaJtvJmSMFuEOSAzw9Hde
b6yhdTynCRmu-kqtO5Dec4lT2OMZKpnxc_F1_4yDJFcqb5CiDSmA-psB2k0Jtj
xAj4UPI61oONK7zzFIu4gBfjJCndsZfdvG7h8wGjV98QhrKEnR7xKZ3KCr0_qR
1B-gxpNk3xWU
.
DKW7jrb4WaRSNfbXVPlT5g
```

Figure 148: Compact Serialization

The resulting JWE object using the JSON General Serialization:

```
{
  "recipients": [
    {
      "encrypted_key": "lJf3HbOApxMEBkCMOoTnnABxs_CvTWUmZQ2ElL
          vYNok"
    }
  ],
  "protected": "eyJhbGciOiJBMjU2R0NNS1ciLCJraWQiOiIxOGVjMDhlMS
      1iZmE5LTRkOTUtYjIwNS0yYjRkZDFkNDMyMWQiLCJ0YWciOiJrZlBkdV
      ZRM1QzSDZ2bmV3dC0ta3N3IiwiaXYiOiJLa1lUMEdYXzJqSGxmcU5fIi
      wiZW5jIjoiQTEyOENCQy1IUzI1NiJ9",
  "iv": "gz6NjyEFNm_vm8Gj6FwoFQ",
  "ciphertext": "Jf5p9-ZhJlJy_IQ_byKFmI0Ro7w7G1QiaZpI8OaiVgD8E
      qoDZHyFKFBupS8iaEeVIgMqWmsuJKuoVgzR3YfzoMd3GxEm3VxNhzWyW
      tZKX0gxKdy6HgLvqoGNbZCzLjqcpDiF8q2_62EVAbr2uSc2oaxFmFuIQ
      HLcqAHxy51449xkjZ7ewzZaGV3eFqhpco8o4DijXaG5_7kp3h2cajRfD
      gymuxUbWgLqaeNQaJtvJmSMFuEOSAzw9Hdeb6yhdTynCRmu-kqtO5Dec
      4lT2OMZKpnxc_F1_4yDJFcqb5CiDSmA-psB2k0JtjxAj4UPI61oONK7z
      zFIu4gBfjJCndsZfdvG7h8wGjV98QhrKEnR7xKZ3KCr0_qR1B-gxpNk3
      xWU",
  "tag": "DKW7jrb4WaRSNfbXVPlT5g"
}
```

Figure 149: JSON General Serialization

The resulting JWE object using the JSON Flattened Serialization:


```
{
  "protected": "eyJhbGciOiJBMjU2R0NNS1ciLCJpdiI6IktrWVQwR1hfMm
      pIbGZxTl8iLCJraWQiOiIxOGVjMDhlMS1iZmE5LTRkOTUtYjIwNS0yYj
      RkZDFkNDMyMWQiLCJ0YWciOiJrZlBkdVZRM1QzSDZ2bmV3dC0ta3N3Ii
      wiZW5jIjoiQTEyOENCQy1IUzI1NiJ9",
  "encrypted_key": "lJf3HbOApxMEBkCMOoTnnABxs_CvTWUmZQ2ElLvYNo
      k",
  "iv": "gz6NjyEFNm_vm8Gj6FwoFQ",
  "ciphertext": "Jf5p9-ZhJlJy_IQ_byKFmI0Ro7w7G1QiaZpI8OaiVgD8E
      qoDZHyFKFBupS8iaEeVIgMqWmsuJKuoVgzR3YfzoMd3GxEm3VxNhzWyW
      tZKX0gxKdy6HgLvqoGNbZCzLjqcpDiF8q2_62EVAbr2uSc2oaxFmFuIQ
      HLcqAHxy51449xkjZ7ewzZaGV3eFqhpco8o4DijXaG5_7kp3h2cajRfD
      gymuxUbWgLqaeNQaJtvJmSMFuEOSAzw9Hdeb6yhdTynCRmu-kqtO5Dec
      4lT2OMZKpnxc_F1_4yDJFcqb5CiDSmA-psB2k0JtjxAj4UPI61oONK7z
      zFIu4gBfjJCndsZfdvG7h8wGjV98QhrKEnR7xKZ3KCr0_qR1B-gxpNk3
      xWU",
  "tag": "NvBveHr_vonkvflfnUrmBQ"
}
```


Figure 150: JSON Flattened Serialization

5.8. Key Wrap using AES-KeyWrap with AES-GCM

The following example illustrates content encryption using the "A128KW" (AES-128-KeyWrap) key encryption algorithm and the "A128GCM" (AES-128-GCM) content encryption algorithm.

Note that whitespace is added for readability as described in Section 1.1.

5.8.1. Input Factors

The following are supplied before beginning the encryption process:

- Plaintext content; this example uses the content from Figure 72.
- AES symmetric key; this example uses the key from Figure 151.
- "alg" parameter of "A128KW"
- "enc" parameter of "A128GCM"

```
{
  "kty": "oct",
  "kid": "81b20965-8332-43d9-a468-82160ad91ac8",
  "use": "enc",
  "alg": "A128KW",
  "k": "GZy6sIZ6wl9NJOKB-jnmVQ"
}
```

Figure 151: AES 128-Bit Key

5.8.2. Generated Factors

The following are generated before encrypting:

- AES symmetric key as the Content Encryption Key; this example uses the key from Figure 152.
- Initialization vector/nonce; this example uses the initialization vector/nonce from Figure 153.

```
aY5_Ghmk9KxWPBLu_glx1w
```

Figure 152: Content Encryption Key, base64url-encoded

```
Qx0pmsDa8KnJc9Jo
```

Figure 153: Initialization Vector, base64url-encoded

5.8.3. Encrypting the Key

Performing the key encryption operation over the CEK (Figure 152) with the AES key (Figure 151) produces the following encrypted key:

```
CBI6oDw8MydIx1IBntf_lQcw2MmJKIQx
```

Figure 154: Encrypted Key, base64url-encoded

5.8.4. Encrypting the Content

The following are generated before encrypting the content:

- JWE Protected Header; this example uses the header from Figure 155, encoded to [RFC4648] base64url as Figure 156.

```
{
  "alg": "A128KW",
  "kid": "81b20965-8332-43d9-a468-82160ad91ac8",
  "enc": "A128GCM"
}
```

Figure 155: JWE Protected Header JSON

```
eyJhbGciOiJBMTI4S1ciLCJraWQiOiI4MWIyMDk2NS04MzMyLTQzZDktYTQ2OC
04MjE2MGFkOTFhYzgiLCJlbmMiOiJBMTI4R0NNIn0
```

Figure 156: JWE Protected Header, base64url-encoded

Performing the content encryption over the Plaintext (Figure 72) with the following:

- CEK (Figure 152);
- Initialization vector/nonce (Figure 153); and
- JWE Protected Header (Figure 156) as authenticated data

produces the following:

- Ciphertext from Figure 157.
- Authentication tag from Figure 158.

```
AwliP-KmWgsZ37BvzCefNen6VTbRK3QMA4TkvRkH0tP1bTdhtFJgJxeVmJkLD6
1A1hnWGetdg11c9ADsnWgL56NyxwSYjU1ZEHcGkd3EkU0vjHi9gTlb90qSYFfe
F0LwkcTtjbYKCsiNJQkcIp1yeM03OmuiYSoYJVSpf7ej6zaYcMv3WwdxDFl8RE
wOhNImk2Xld2JXq6BR53TSFkyT7PwVLuq-1GwtGHlQeg7gDT6xW0JqHDPn_H-p
uQsmthc9Zg0ojmJfqqFvETUxLAF-KjcBTS5dNy6egwkYtOt8EIHK-oEsKYtZRa
a8Z7MOZ7UGxGIMvEmxrGCPeJa14slv2-gaqK0kEThkaSqdYw0FkQZF
```

Figure 157: Ciphertext, base64url-encoded

And authentication tag:

```
ER7MWJZ1FBI_NKvn7Zb1Lw
```

Figure 158: Authentication Tag, base64url-encoded

5.8.5. Output Results

The following compose the resulting JWE object:

- JWE Protected Header (Figure 156)

- encrypted key (Figure 154)
- Initialization vector/nonce (Figure 153)
- Ciphertext (Figure 157)
- Authentication tag (Figure 158)

The resulting JWE object using the Compact serialization:

```
eyJhbGciOiJBMTI4S1ciLCJraWQiOiI4MWIyMDk2NS04MzMyLTQzZDktYTQ2OC
04MjE2MGFkOTFhYzgiLCJlbmMiOiJBMTI4R0NNIn0
.
CBI6oDw8MydIx1IBntf_lQcw2MmJKIQx
.
Qx0pmsDa8KnJc9Jo
.
AwliP-KmWgsZ37BvzCefNen6VTbRK3QMA4TkvRkH0tP1bTdhtFJgJxeVmJkLD6
1A1hnWGetdg11c9ADsnWgL56NyxwSYjU1ZEHcGkd3EkU0vjHi9gTlb90qSYFfe
F0LwkcTtjbYKCsiNJQkcIp1yeM03OmuiYSoYJVSpf7ej6zaYcMv3WwdxDFl8RE
wOhNImk2Xld2JXq6BR53TSFkyT7PwVLuq-1GwtGHlQeg7gDT6xW0JqHDPn_H-p
uQsmthc9Zg0ojmJfqqFvETUxLAF-KjcBTS5dNy6egwkYtOt8EIHK-oEsKYtZRa
a8Z7MOZ7UGxGIMvEmxrGCPeJa14slv2-gaqK0kEThkaSqdYw0FkQZF
.
ER7MWJZ1FBI_NKvn7Zb1Lw
```

Figure 159: Compact Serialization

The resulting JWE object using the JSON General Serialization:

```
{
  "recipients": [
    {
      "encrypted_key": "CBI6oDw8MydIx1IBntf_lQcw2MmJKIQx"
    }
  ],
  "protected": "eyJhbGciOiJBMTI4S1ciLCJraWQiOiI4MWIyMDk2NS04Mz
      MyLTQzZDktYTQ2OC04MjE2MGFkOTFhYzgiLCJlbmMiOiJBMTI4R0NNIn
      0",
  "iv": "Qx0pmsDa8KnJc9Jo",
  "ciphertext": "AwliP-KmWgsZ37BvzCefNen6VTbRK3QMA4TkvRkH0tP1b
      TdhtFJgJxeVmJkLD61A1hnWGetdg11c9ADsnWgL56NyxwSYjU1ZEHcGk
      d3EkU0vjHi9gTlb90qSYFfeF0LwkcTtjbYKCsiNJQkcIp1yeM03OmuiY
      SoYJVSpf7ej6zaYcMv3WwdxDFl8REwOhNImk2Xld2JXq6BR53TSFkyT7
      PwVLuq-1GwtGHlQeg7gDT6xW0JqHDPn_H-puQsmthc9Zg0ojmJfqqFvE
      TUxLAF-KjcBTS5dNy6egwkYtOt8EIHK-oEsKYtZRaa8Z7MOZ7UGxGIMv
      EmxrGCPeJa14slv2-gaqK0kEThkaSqdYw0FkQZF",
  "tag": "ER7MWJZ1FBI_NKvn7Zb1Lw"
}
```


Figure 160: JSON General Serialization

The resulting JWE object using the JSON Flattened Serialization:


```
{
  "protected": "eyJhbGciOiJBMTI4S1ciLCJraWQiOiI4MWIyMDk2NS04Mz
      MyLTQzZDktYTQ2OC04MjE2MGFkOTFhYzgiLCJlbmMiOiJBMTI4R0NNIn
      0",
  "encrypted_key": "CBI6oDw8MydIx1IBntf_lQcw2MmJKIQx",
  "iv": "Qx0pmsDa8KnJc9Jo",
  "ciphertext": "AwliP-KmWgsZ37BvzCefNen6VTbRK3QMA4TkvRkH0tP1b
      TdhtFJgJxeVmJkLD61A1hnWGetdg11c9ADsnWgL56NyxwSYjU1ZEHcGk
      d3EkU0vjHi9gTlb90qSYFfeF0LwkcTtjbYKCsiNJQkcIp1yeM03OmuiY
      SoYJVSpf7ej6zaYcMv3WwdxDFl8REwOhNImk2Xld2JXq6BR53TSFkyT7
      PwVLuq-1GwtGHlQeg7gDT6xW0JqHDPn_H-puQsmthc9Zg0ojmJfqqFvE
      TUxLAF-KjcBTS5dNy6egwkYtOt8EIHK-oEsKYtZRaa8Z7MOZ7UGxGIMv
      EmxrGCPeJa14slv2-gaqK0kEThkaSqdYw0FkQZF",
  "tag": "ER7MWJZ1FBI_NKvn7Zb1Lw"
}
```


Figure 161: JSON Flattened Serialization

5.9. Compressed Content

This example illustrates encrypting content that is first compressed. It reuses the AES key, key encryption algorithm, and content encryption algorithm from Section 5.8.

Note that whitespace is added for readability as described in Section 1.1.

5.9.1. Input Factors

The following are supplied before beginning the encryption process:

- Plaintext content; this example uses the content from Figure 72.
- Recipient encryption key; this example uses the key from Figure 151.
- Key encryption algorithm; this example uses "A128KW".
- Content encryption algorithm; this example uses "A128GCM".
- "zip" parameter as "DEF".

5.9.2. Generated Factors

The following are generated before encrypting:

- Compressed plaintext from the original plaintext content; compressing Figure 72 using the DEFLATE [RFC1951] algorithm produces the compressed plaintext from Figure 162.
- AES symmetric key as the Content Encryption Key (CEK); this example uses the key from Figure 163.
- Initialization vector/nonce; this example uses the initialization vector/nonce from Figure 164.

```
bY_BDcIwDEVX-QNU3QEOrIA4pqlDokYxchxVvbEDGzIJbioOSJwc-f___HPjBu
8KVFpVtAplVE1-wZo0YjNZo3C7R5v72pV5f5X382VWjYQpqZKAyjziZOr2B7kQ
PSy6oZIXUnDYbVKN4jNXi2u0yB7t1qSHTjmMODf9QgvrDzfTIQXnyQRuUya4zI
WG3vTOdir0v7BRHFYWq3k1k1A_gSDJqtcBF-GZxw8
```

Figure 162: Compressed Plaintext, base64url-encoded

```
hC-MpLZSuwWv8sexS6ydfw
```

Figure 163: Content Encryption Key, base64url-encoded

```
p9pUq6XHY0jfEZIl
```

Figure 164: Initialization Vector, base64url-encoded

5.9.3. Encrypting the Key

Performing the key encryption operation over the CEK (Figure 163) with the AES key (Figure 151) produces the following encrypted key:

```
5vUT2WOtQxKWcekM_IzVQwkGgzlFDwPi
```

Figure 165: Encrypted Key, base64url-encoded

5.9.4. Encrypting the Content

The following are generated before encrypting the content:

- JWE Protected Header; this example uses the header from Figure 166, encoded as [RFC4648] base64url as Figure 167.

```
{
  "alg": "A128KW",
  "kid": "81b20965-8332-43d9-a468-82160ad91ac8",
  "enc": "A128GCM",
  "zip": "DEF"
}
```

Figure 166: JWE Protected Header JSON

```
eyJhbGciOiJBMTI4S1ciLCJraWQiOiI4MWIyMDk2NS04MzMyLTQzZDktYTQ2OC
04MjE2MGFkOTFhYzgiLCJlbmMiOiJBMTI4R0NNIiwiemlwIjoiREVGIn0
```

Figure 167: JWE Protected Header, base64url-encoded

Performing the content encryption operation over the compressed Plaintext (Figure 162, encoded as an octet string) with the following:

- CEK (Figure 163);
- Initialization vector/nonce (Figure 164); and
- JWE Protected Header (Figure 167) as authenticated data

produces the following:

- Ciphertext from Figure 168.
- Authentication tag from Figure 169.

```
HbDtOsdai1oYziSx25KEeTxmwnh8L8jKMFNc1k3zmMI6VB8hry57tDZ61jXyez
SPt0fdLVfe6Jf5y5-JaCap_JQBcb5opbmT60uWGml8blyiMQmOn9J--XhhlYg0
m-BHaqfDO5iTOWxPxFMUedx7WCy8mxgDHj0aBMG6152PsM-w5E_o2B3jDbrYBK
hpYA7qi3AyijnCJ7BP9rr3U8kxExCpG3mK420TjOw
```

Figure 168: Ciphertext, base64url-encoded

And authentication tag:

```
VILuUwuIxaLVmh5X-T7kmA
```

Figure 169: Authentication Tag, base64url-encoded

5.9.5. Output Results

The following compose the resulting JWE object:

- JWE Protected Header (Figure 167)
- encrypted key (Figure 165)
- Initialization vector/nonce (Figure 164)
- Ciphertext (Figure 168)
- Authentication tag (Figure 169)

The resulting JWE object using the Compact serialization:

```
eyJhbGciOiJBMTI4S1ciLCJraWQiOiI4MWIyMDk2NS04MzMyLTQzZDktYTQ2OC
04MjE2MGFkOTFhYzgiLCJlbmMiOiJBMTI4R0NNIiwiemlwIjoiREVGIn0
.
5vUT2WOtQxKWcekM_IzVQwkGgzlFDwPi
.
p9pUq6XHY0jfEZIl
.
HbDtOsdai1oYziSx25KEeTxmwnh8L8jKMFNc1k3zmMI6VB8hry57tDZ61jXyez
SPt0fdLVfe6Jf5y5-JaCap_JQBcb5opbmT60uWGml8blyiMQmOn9J--XhhlYg0
m-BHaqfDO5iTOWxPxFMUedx7WCy8mxgDHj0aBMG6152PsM-w5E_o2B3jDbrYBK
hpYA7qi3AyijnCJ7BP9rr3U8kxExCpG3mK420TjOw
.
VILuUwuIxaLVmh5X-T7kmA
```

Figure 170: Compact Serialization

The resulting JWE object using the JSON General Serialization:

```
{
  "recipients": [
    {
      "encrypted_key": "5vUT2WOtQxKWcekM_IzVQwkGgzlFDwPi"
    }
  ],
  "protected": "eyJhbGciOiJBMTI4S1ciLCJraWQiOiI4MWIyMDk2NS04Mz
      MyLTQzZDktYTQ2OC04MjE2MGFkOTFhYzgiLCJlbmMiOiJBMTI4R0NNIi
      wiemlwIjoiREVGIn0",
  "iv": "p9pUq6XHY0jfEZIl",
  "ciphertext": "HbDtOsdai1oYziSx25KEeTxmwnh8L8jKMFNc1k3zmMI6V
      B8hry57tDZ61jXyezSPt0fdLVfe6Jf5y5-JaCap_JQBcb5opbmT60uWG
      ml8blyiMQmOn9J--XhhlYg0m-BHaqfDO5iTOWxPxFMUedx7WCy8mxgDH
      j0aBMG6152PsM-w5E_o2B3jDbrYBKhpYA7qi3AyijnCJ7BP9rr3U8kxE
      xCpG3mK420TjOw",
  "tag": "VILuUwuIxaLVmh5X-T7kmA"
}
```

Figure 171: JSON General Serialization

The resulting JWE object using the JSON Flattened Serialization:

```
{
  "protected": "eyJhbGciOiJBMTI4S1ciLCJraWQiOiI4MWIyMDk2NS04Mz
      MyLTQzZDktYTQ2OC04MjE2MGFkOTFhYzgiLCJlbmMiOiJBMTI4R0NNIi
      wiemlwIjoiREVGIn0",
  "encrypted_key": "5vUT2WOtQxKWcekM_IzVQwkGgzlFDwPi",
  "iv": "p9pUq6XHY0jfEZIl",
  "ciphertext": "HbDtOsdai1oYziSx25KEeTxmwnh8L8jKMFNc1k3zmMI6V
      B8hry57tDZ61jXyezSPt0fdLVfe6Jf5y5-JaCap_JQBcb5opbmT60uWG
      ml8blyiMQmOn9J--XhhlYg0m-BHaqfDO5iTOWxPxFMUedx7WCy8mxgDH
      j0aBMG6152PsM-w5E_o2B3jDbrYBKhpYA7qi3AyijnCJ7BP9rr3U8kxE
      xCpG3mK420TjOw",
  "tag": "VILuUwuIxaLVmh5X-T7kmA"
}
```

Figure 172: JSON Flattened Serialization

5.10. Including Additional Authenticated Data

This example illustrates encrypting content that includes additional authenticated data. As this example includes an additional top-level property not present in the Compact serialization, only the JSON serialization is possible.

Note that whitespace is added for readability as described in Section 1.1.

5.10.1. Input Factors

The following are supplied before beginning the encryption process:

- Plaintext content; this example uses the content from Figure 72.
- Recipient encryption key; this example uses the key from Figure 151.
- Key encryption algorithm; this example uses "A128KW".
- Content encryption algorithm; this example uses "A128GCM".
- Additional authenticated data; this example uses a [RFC7095] vCard from Figure 173, serialized to UTF-8.

```
[
  "vcard",
  [
    [ "version", {}, "text", "4.0" ],
    [ "fn", {}, "text", "Meriadoc Brandybuck" ],
    [ "n", {},
      "text", [
        "Brandybuck", "Meriadoc", "Mr.", ""
      ]
    ],
    [ "bday", {}, "text", "TA 2982" ],
    [ "gender", {}, "text", "M" ]
  ]
]
```

Figure 173: Additional Authenticated Data, in JSON format

NOTE whitespace between JSON values added for readability.

5.10.2. Generated Factors

The following are generated before encrypting:

- AES symmetric key as the Content Encryption Key (CEK); this example uses the key from Figure 174.
- Initialization vector/nonce; this example uses the initialization vector/nonce from Figure 175.
- Encoded additional authenticated data (AAD); this example uses the additional authenticated data from Figure 173, encoded to [RFC4648] base64url as Figure 176.

```
75m1ALsYv10pZTKPWrsqdg
```

Figure 174: Content Encryption Key, base64url-encoded

```
veCx9ece2orS7c_N
```

Figure 175: Initialization Vector, base64url-encoded

```
WyJ2Y2FyZCIsW1sidmVyc2lvbiIse30sInRleHQiLCI0LjAiXSxbImZuIix7fS
widGV4dCIsIk1lcmlhZG9jIEJyYW5keWJ1Y2siXSxbIm4iLHt9LCJ0ZXh0Iixb
IkJyYW5keWJ1Y2siLCJNZXJpYWRvYyIsIk1yLiIsIiJdXSxbImJkYXkiLHt9LC
J0ZXh0IiwiVEEgMjk4MiJdLFsiZ2VuZGVyIix7fSwidGV4dCIsIk0iXV1d
```

Figure 176: Additional Authenticated Data, base64url-encoded

5.10.3. Encrypting the Key

Performing the key encryption operation over the CEK (Figure 174) with the AES key (Figure 151) produces the following encrypted key:

```
4YiiQ_ZzH76TaIkJmYfRFgOV9MIpnx4X
```

Figure 177: Encrypted Key, base64url-encoded

5.10.4. Encrypting the Content

The following are generated before encrypting the content:

- JWE Protected Header; this example uses the header from Figure 178, encoded to [RFC4648] base64url as Figure 179.

```
{
  "alg": "A128KW",
  "kid": "81b20965-8332-43d9-a468-82160ad91ac8",
  "enc": "A128GCM"
}
```

Figure 178: JWE Protected Header JSON

```
eyJhbGciOiJBMTI4S1ciLCJraWQiOiI4MWIyMDk2NS04MzMyLTQzZDktYTQ2OC
04MjE2MGFkOTFhYzgiLCJlbmMiOiJBMTI4R0NNIn0
```

Figure 179: JWE Protected Header, base64url-encoded

Performing the content encryption operation over the Plaintext with the following:

- CEK (Figure 174);

- Initialization vector/nonce (Figure 175); and
- Concatenation of the JWE Protected Header (Figure 179), ".", and the [RFC4648] base64url encoding of Figure 173 as authenticated data

produces the following:

- Ciphertext from Figure 180.
- Authentication tag from Figure 181.

```
Z_3cbr0k3bVM6N3oSNmHz7Lyf3iPppGf3Pj17wNZqteJ0Ui8p74SchQP8xygM1
oFRWCNzeIa6s6BcEtp8qEFiqTUEyiNkOWDNoF14T_4NFqF-p2Mx8zkbKxI7oPK
8KNarFbyxIDvICNqBLba-v3uzXBdB89fzOI-Lv4PjOFAQGHrgv1rjXAmKbgkft
9cB4WeyZw8MldbBhc-V_KWZslrsLNygon_JJWd_ek6LQn5NRehvApqf9ZrxB4a
q3FXBxOxCys35PhCdaggy2kfUfl2OkwKnWUbgXVD1C6HxLIlqHhCwXDG59weHr
RDQeHyMRoBljoV3X_bUTJDnKBFOod7nLz-cj48JMx3SnCZTpbQAkFV
```

Figure 180: Ciphertext, base64url-encoded

```
vOaH_Rajnpy_3hOtqvZHRA
```

Figure 181: Authentication Tag, base64url-encoded

5.10.5. Output Results

The following compose the resulting JWE object:

- JWE Protected Header (Figure 179)
- encrypted key (Figure 177)
- Initialization vector/nonce (Figure 175)
- Additional authenticated data (Figure 176)
- Ciphertext (Figure 180)
- Authentication tag (Figure 181)

The Compact Serialization is not presented because it does not support this use case.

The resulting JWE object using the JSON General Serialization:

```
{
  "recipients": [
    {
      "encrypted_key": "4YiiQ_ZzH76TaIkJmYfRFgOV9MIpnx4X"
    }
  ],
  "protected": "eyJhbGciOiJBMTI4S1ciLCJraWQiOiI4MWIyMDk2NS04Mz
      MyLTQzZDktYTQ2OC04MjE2MGFkOTFhYzgiLCJlbmMiOiJBMTI4R0NNIn
      0",
  "iv": "veCx9ece2orS7c_N",
  "aad": "WyJ2Y2FyZCIsW1sidmVyc2lvbiIse30sInRleHQiLCI0LjAiXSxb
      ImZuIix7fSwidGV4dCIsIk1lcmlhZG9jIEJyYW5keWJ1Y2siXSxbIm4i
      LHt9LCJ0ZXh0IixbIkJyYW5keWJ1Y2siLCJNZXJpYWRvYyIsIk1yLiIs
      IiJdXSxbImJkYXkiLHt9LCJ0ZXh0IiwiVEEgMjk4MiJdLFsiZ2VuZGVy
      Iix7fSwidGV4dCIsIk0iXV1d",
  "ciphertext": "Z_3cbr0k3bVM6N3oSNmHz7Lyf3iPppGf3Pj17wNZqteJ0
      Ui8p74SchQP8xygM1oFRWCNzeIa6s6BcEtp8qEFiqTUEyiNkOWDNoF14
      T_4NFqF-p2Mx8zkbKxI7oPK8KNarFbyxIDvICNqBLba-v3uzXBdB89fz
      OI-Lv4PjOFAQGHrgv1rjXAmKbgkft9cB4WeyZw8MldbBhc-V_KWZslrs
      LNygon_JJWd_ek6LQn5NRehvApqf9ZrxB4aq3FXBxOxCys35PhCdaggy
      2kfUfl2OkwKnWUbgXVD1C6HxLIlqHhCwXDG59weHrRDQeHyMRoBljoV3
      X_bUTJDnKBFOod7nLz-cj48JMx3SnCZTpbQAkFV",
  "tag": "vOaH_Rajnpy_3hOtqvZHRA"
}
```


Figure 182: JSON General Serialization

The resulting JWE object using the JSON Flattened Serialization:

```
{
  "protected": "eyJhbGciOiJBMTI4S1ciLCJraWQiOiI4MWIyMDk2NS04Mz
      MyLTQzZDktYTQ2OC04MjE2MGFkOTFhYzgiLCJlbmMiOiJBMTI4R0NNIn
      0",
  "encrypted_key": "4YiiQ_ZzH76TaIkJmYfRFgOV9MIpnx4X",
  "aad": "WyJ2Y2FyZCIsW1sidmVyc2lvbiIse30sInRleHQiLCI0LjAiXSxb
      ImZuIix7fSwidGV4dCIsIk1lcmlhZG9jIEJyYW5keWJ1Y2siXSxbIm4i
      LHt9LCJ0ZXh0IixbIkJyYW5keWJ1Y2siLCJNZXJpYWRvYyIsIk1yLiIs
      IiJdXSxbImJkYXkiLHt9LCJ0ZXh0IiwiVEEgMjk4MiJdLFsiZ2VuZGVy
      Iix7fSwidGV4dCIsIk0iXV1d",
  "iv": "veCx9ece2orS7c_N",
  "ciphertext": "Z_3cbr0k3bVM6N3oSNmHz7Lyf3iPppGf3Pj17wNZqteJ0
      Ui8p74SchQP8xygM1oFRWCNzeIa6s6BcEtp8qEFiqTUEyiNkOWDNoF14
      T_4NFqF-p2Mx8zkbKxI7oPK8KNarFbyxIDvICNqBLba-v3uzXBdB89fz
      OI-Lv4PjOFAQGHrgv1rjXAmKbgkft9cB4WeyZw8MldbBhc-V_KWZslrs
      LNygon_JJWd_ek6LQn5NRehvApqf9ZrxB4aq3FXBxOxCys35PhCdaggy
      2kfUfl2OkwKnWUbgXVD1C6HxLIlqHhCwXDG59weHrRDQeHyMRoBljoV3
      X_bUTJDnKBFOod7nLz-cj48JMx3SnCZTpbQAkFV",
  "tag": "vOaH_Rajnpy_3hOtqvZHRA"
}
```


Figure 183: JSON Flattened Serialization

5.11. Protecting Specific Header Fields

This example illustrates encrypting content where only certain JOSE header parameters are protected. As this example includes parameters in the JWE Shared Unprotected Header, only the JSON General Serialization and JSON Flattened Serialization are possible.

Note that whitespace is added for readability as described in Section 1.1.

5.11.1. Input Factors

The following are supplied before beginning the encryption process:

- Plaintext content; this example uses the content from Figure 72.
- Recipient encryption key; this example uses the key from Figure 151.
- Key encryption algorithm; this example uses "A128KW".
- Content encryption algorithm; this example uses "A128GCM".

5.11.2. Generated Factors

The following are generated before encrypting:

- AES symmetric key as the Content Encryption Key (CEK); this example uses the key from Figure 184.
- Initialization vector/nonce; this example uses the initialization vector/nonce from Figure 185.

```
WDgEptBmQs9ouUvArz6x6g
```

Figure 184: Content Encryption Key, base64url-encoded

```
WgEJsDS9bkoXQ3nR
```

Figure 185: Initialization Vector, base64url-encoded

5.11.3. Encrypting the Key

Performing the key encryption operation over the CEK (Figure 184) with the AES key (Figure 151) produces the following encrypted key:

```
jJIcM9J-hbx3wnqhf5FlkEYos0sHsF0H
```

Figure 186: Encrypted Key, base64url-encoded

5.11.4. Encrypting the Content

The following are generated before encrypting the content:

- JWE Protected Header; this example uses the header from Figure 187, encoded to [RFC4648] base64url as Figure 188.

```
{
  "enc": "A128GCM"
}
```

Figure 187: JWE Protected Header JSON

```
eyJlbmMiOiJBMTI4R0NNIn0
```

Figure 188: JWE Protected Header, base64url-encoded

Performing the content encryption operation over the Plaintext with the following:

- CEK (Figure 184);

- Initialization vector/nonce (Figure 185); and
- JWE Protected Header (Figure 188) as authenticated data

produces the following:

- Ciphertext from Figure 189.
- Authentication tag from Figure 190.

```
lIbCyRmRJxnB2yLQOTqjCDKV3H30ossOw3uD9DPsqLL2DM3swKkjOwQyZtWsFL
YMj5YeLht_StAn21tHmQJuuNt64T8D4t6C7kC9OCCJ1IHAolUv4MyOt80MoPb8
fZYbNKqplzYJgIL58g8N2v46OgyG637d6uuKPwhAnTGm_zWhqc_srOvgiLkzyF
XPq1hBAURbc3-8BqeRb48iR1-_5g5UjWVD3lgiLCN_P7AW8mIiFvUNXBPJK3nO
WL4teUPS8yHLbWeL83olU4UAgL48x-8dDkH23JykibVSQju-f7e-1xreHWXzWL
Hs1NqBbre0dEwK3HX_xM0LjUz77Krppgegoutpf5qaKg3l-_xMINmf
```

Figure 189: Ciphertext, base64url-encoded

```
fNYLqpUe84KD45lvDiaBAQ
```

Figure 190: Authentication Tag, base64url-encoded

5.11.5. Output Results

The following compose the resulting JWE object:

- JWE Shared Unprotected Header (Figure 191)
- JWE Protected Header (Figure 188)
- encrypted key (Figure 186)
- Initialization vector/nonce (Figure 185)
- Ciphertext (Figure 189)
- Authentication tag (Figure 190)

The Compact Serialization is not presented because it does not support this use case.

The following JWE Shared Unprotected Header is generated before assembling the output results:

```
{
  "alg": "A128KW",
  "kid": "81b20965-8332-43d9-a468-82160ad91ac8"
}
```

Figure 191: JWE Shared Unprotected Header JSON

The resulting JWE object using the JSON General Serialization:

```
{
  "recipients": [
    {
      "encrypted_key": "jJIcM9J-hbx3wnqhf5FlkEYos0sHsF0H"
    }
  ],
  "unprotected": {
    "alg": "A128KW",
    "kid": "81b20965-8332-43d9-a468-82160ad91ac8"
  },
  "protected": "eyJlbmMiOiJBMTI4R0NNIn0",
  "iv": "WgEJsDS9bkoXQ3nR",
  "ciphertext": "lIbCyRmRJxnB2yLQOTqjCDKV3H30ossOw3uD9DPsqLL2D
      M3swKkjOwQyZtWsFLYMj5YeLht_StAn21tHmQJuuNt64T8D4t6C7kC9O
      CCJ1IHAolUv4MyOt80MoPb8fZYbNKqplzYJgIL58g8N2v46OgyG637d6
      uuKPwhAnTGm_zWhqc_srOvgiLkzyFXPq1hBAURbc3-8BqeRb48iR1-_5
      g5UjWVD3lgiLCN_P7AW8mIiFvUNXBPJK3nOWL4teUPS8yHLbWeL83olU
      4UAgL48x-8dDkH23JykibVSQju-f7e-1xreHWXzWLHs1NqBbre0dEwK3
      HX_xM0LjUz77Krppgegoutpf5qaKg3l-_xMINmf",
  "tag": "fNYLqpUe84KD45lvDiaBAQ"
}
```

Figure 192: JSON General Serialization

The resulting JWE object using the JSON Flattened Serialization:

```
{
  "protected": "eyJlbmMiOiJBMTI4R0NNIn0",
  "unprotected": {
    "alg": "A128KW",
    "kid": "81b20965-8332-43d9-a468-82160ad91ac8"
  },
  "encrypted_key": "jJIcM9J-hbx3wnqhf5FlkEYos0sHsF0H",
  "iv": "WgEJsDS9bkoXQ3nR",
  "ciphertext": "lIbCyRmRJxnB2yLQOTqjCDKV3H30ossOw3uD9DPsqLL2D
      M3swKkjOwQyZtWsFLYMj5YeLht_StAn21tHmQJuuNt64T8D4t6C7kC9O
      CCJ1IHAolUv4MyOt80MoPb8fZYbNKqplzYJgIL58g8N2v46OgyG637d6
      uuKPwhAnTGm_zWhqc_srOvgiLkzyFXPq1hBAURbc3-8BqeRb48iR1-_5
      g5UjWVD3lgiLCN_P7AW8mIiFvUNXBPJK3nOWL4teUPS8yHLbWeL83olU
      4UAgL48x-8dDkH23JykibVSQju-f7e-1xreHWXzWLHs1NqBbre0dEwK3
      HX_xM0LjUz77Krppgegoutpf5qaKg3l-_xMINmf",
  "tag": "fNYLqpUe84KD45lvDiaBAQ"
}
```

Figure 193: JSON Flattened Serialization

5.12. Protecting Content Only

This example illustrates encrypting content where none of the JOSE header parameters are protected. As this example includes parameters only in the JWE Shared Unprotected Header, only the JSON serialization is possible.

Note that whitespace is added for readability as described in Section 1.1.

5.12.1. Input Factors

The following are supplied before beginning the encryption process:

- Plaintext content; this example uses the content from Figure 72.
- Recipient encryption key; this example uses the key from Figure 151.
- Key encryption algorithm; this example uses "A128KW".
- Content encryption algorithm; this example uses "A128GCM".

5.12.2. Generated Factors

The following are generated before encrypting:

- o AES symmetric key as the Content Encryption Key; this example the key from Figure 194.

- o Initialization vector/nonce; this example uses the initialization vector/nonce from Figure 195.

```
KBooAFl30QPV3vkcZlXnzQ
```

Figure 194: Content Encryption Key, base64url-encoded

```
YihBoVOGsR1l7jCD
```

Figure 195: Initialization Vector, base64url-encoded

5.12.3. Encrypting the Key

Performing the key encryption operation over the CEK (Figure 194 with the AES key (Figure 151 produces the following encrypted key:

```
244YHfO_W7RMpQW81UjQrZcq5LSyqiPv
```

Figure 196: Encrypted Key, base64url-encoded

5.12.4. Encrypting the Content

Performing the content encryption operation over the Plaintext (Figure 72) using the following:

- o CEK (Figure 194);

- o Initialization vector/nonce (Figure 195); and

- o Empty string as authenticated data

produces the following:

- o Ciphertext from Figure 197.

- o Authenticated data from Figure 198.

```
qtPIMMaOBRgASL10dNQhOa7Gqrk7Eal1vwht7R4TT1uq-arsVCPaIeFwQfzrSS
6oEUWbBtxEasE0vC6r7sphyVziMCVJEuRJyoAHFSP3eqQPb4Ic1SDSqyXjw_L3
svybhHYUGyQuTmUQEDjgjJfBOifwHIsDsRPeBz1NomqeifVPq5GTCWFo5k_MNI
QURR2Wj0AHC2k7JZfu2iWjUHLF8ExFZLZ4nlmsvJu_mvifMYiikfNfsZAudISO
a6O73yPZtL04k_1FI7WDfrb2w7OqKLWDXzlpcxohPVOLQwpA3mFNRKdY-bQz4Z
4KX9lfz1cne31N4-8BKmojpw-OdQjKdLOGkC445Fb_K1tlDQXw2sBF
```

Figure 197: Ciphertext, base64url-encoded

```
e2m0Vm7JvjK2VpCKXS-kyg
```

Figure 198: Authentication Tag, base64url-encoded

5.12.5. Output Results

The Compact Serialization is not presented because it does not support this use case.

The following JWE Shared Unprotected Header is generated before assembling the output results:

```
{
  "alg": "A128KW",
  "kid": "81b20965-8332-43d9-a468-82160ad91ac8",
  "enc": "A128GCM"
}
```

Figure 199: JWE Shared Unprotected Header JSON

The following compose the resulting JWE object:

- JWE Shared Unprotected Header (Figure 199)
- encrypted key (Figure 196)
- Initialization vector/nonce (Figure 195)
- Ciphertext (Figure 197)
- Authentication tag (Figure 198)

The resulting JWE object using the JSON General Serialization:

```
{
  "recipients": [
    {
      "encrypted_key": "244YHfO_W7RMpQW81UjQrZcq5LSyqiPv"
    }
  ],
  "unprotected": {
    "alg": "A128KW",
    "kid": "81b20965-8332-43d9-a468-82160ad91ac8",
    "enc": "A128GCM"
  },
  "iv": "YihBoVOGsR1l7jCD",
  "ciphertext": "qtPIMMaOBRgASL10dNQhOa7Gqrk7Eal1vwht7R4TT1uq-
      arsVCPaIeFwQfzrSS6oEUWbBtxEasE0vC6r7sphyVziMCVJEuRJyoAHF
      SP3eqQPb4Ic1SDSqyXjw_L3svybhHYUGyQuTmUQEDjgjJfBOifwHIsDs
      RPeBz1NomqeifVPq5GTCWFo5k_MNIQURR2Wj0AHC2k7JZfu2iWjUHLF8
      ExFZLZ4nlmsvJu_mvifMYiikfNfsZAudISOa6O73yPZtL04k_1FI7WDf
      rb2w7OqKLWDXzlpcxohPVOLQwpA3mFNRKdY-bQz4Z4KX9lfz1cne31N4
      -8BKmojpw-OdQjKdLOGkC445Fb_K1tlDQXw2sBF",
  "tag": "e2m0Vm7JvjK2VpCKXS-kyg"
}
```

Figure 200: JSON General Serialization

The resulting JWE object using the JSON Flattened Serialization:

```
{
  "unprotected": {
    "alg": "A128KW",
    "kid": "81b20965-8332-43d9-a468-82160ad91ac8",
    "enc": "A128GCM"
  },
  "encrypted_key": "244YHfO_W7RMpQW81UjQrZcq5LSyqiPv",
  "iv": "YihBoVOGsR1l7jCD",
  "ciphertext": "qtPIMMaOBRgASL10dNQhOa7Gqrk7Eal1vwht7R4TT1uq-
      arsVCPaIeFwQfzrSS6oEUWbBtxEasE0vC6r7sphyVziMCVJEuRJyoAHF
      SP3eqQPb4Ic1SDSqyXjw_L3svybhHYUGyQuTmUQEDjgjJfBOifwHIsDs
      RPeBz1NomqeifVPq5GTCWFo5k_MNIQURR2Wj0AHC2k7JZfu2iWjUHLF8
      ExFZLZ4nlmsvJu_mvifMYiikfNfsZAudISOa6O73yPZtL04k_1FI7WDf
      rb2w7OqKLWDXzlpcxohPVOLQwpA3mFNRKdY-bQz4Z4KX9lfz1cne31N4
      -8BKmojpw-OdQjKdLOGkC445Fb_K1tlDQXw2sBF",
  "tag": "e2m0Vm7JvjK2VpCKXS-kyg"
}
```

Figure 201: JSON Flattened Serialization

5.13. Encrypting to Multiple Recipients

This example illustrates encryption content for multiple recipients. As this example has multiple recipients, only the JSON serialization is possible.

Note that RSAES-PKCS1-v1_5 uses random data to generate the ciphertext; it might not be possible to exactly replicate the results in this section.

Note that whitespace is added for readability as described in Section 1.1.

5.13.1. Input Factors

The following are supplied before beginning the encryption process:

- Plaintext content; this example uses the plaintext from Figure 72.
- Recipient keys; this example uses the following:
  - The RSA public key from Figure 73 for the first recipient.
  - The EC public key from Figure 108 for the second recipient.
  - The AES symmetric key from Figure 138 for the third recipient.
- Key encryption algorithms; this example uses the following:
  - "RSA1_5" for the first recipient.
  - "ECDH-ES+A256KW" for the second recipient.
  - "A256GCMKW" for the third recipient.
- Content encryption algorithm; this example uses "A128CBC-HS256"

5.13.2. Generated Factors

The following are generated before encrypting:

- AES symmetric key as the Content Encryption Key (CEK); this example uses the key from Figure 202.
- Initialization vector/nonce; this example uses the initialization vector/nonce from Figure 203.

```
zXayeJ4gvm8NJr3IUInyokTUO-LbQNKEhe_zWlYbdpQ
```

Figure 202: Content Encryption Key, base64url-encoded

```
VgEIHY20EnzUtZFl2RpB1g
```

Figure 203: Initialization Vector, base64url-encoded

5.13.3. Encrypting the Key to the First Recipient

Performing the "RSA1_5" key encryption operation over the CEK (Figure 202 with the first recipient's RSA key (Figure 73 produces the following encrypted key:

```
dYOD28kab0Vvf4ODgxVAJXgHcSZICSOp8M51zjwj4w6Y5G4XJQsNNIBiqyvUUA
OcpL7S7-cFe7Pio7gV_Q06WmCSa-vhW6me4bWrBf7cHwEQJdXihidAYWVajJIa
KMXMvFRMV6iDlRr076DFthg2_AV0_tSiV6xSEIFqt1xnYPpmP91tc5WJDOGb-w
qjw0-b-S1laS11QVbuP78dQ7Fa0zAVzzjHX-xvyM2wxj_otxr9clN1LnZMbeYS
rRicJK5xodvWgkpIdkMHo4LvdhRRvzoKzlic89jFWPlnBq_V4n5trGuExtp_-d
bHcGlihqc_wGgho9fLMK8JOArYLcMDNQ
```

Figure 204: Recipient #1 Encrypted Key, base64url-encoded

The following are generated after encrypting the CEK for the first recipient:

- Recipient JWE Unprotected Header from Figure 205

```
{
  "alg": "RSA1_5",
  "kid": "frodo.baggins@hobbiton.example"
}
```

Figure 205: Recipient #1 JWE Per-recipient Unprotected Header JSON

The following is the assembled first recipient JSON:

```
{
  "encrypted_key": "dYOD28kab0Vvf4ODgxVAJXgHcSZICSOp8M51zjwj4w
      6Y5G4XJQsNNIBiqyvUUAOcpL7S7-cFe7Pio7gV_Q06WmCSa-vhW6me4b
      WrBf7cHwEQJdXihidAYWVajJIaKMXMvFRMV6iDlRr076DFthg2_AV0_t
      SiV6xSEIFqt1xnYPpmP91tc5WJDOGb-wqjw0-b-S1laS11QVbuP78dQ7
      Fa0zAVzzjHX-xvyM2wxj_otxr9clN1LnZMbeYSrRicJK5xodvWgkpIdk
      MHo4LvdhRRvzoKzlic89jFWPlnBq_V4n5trGuExtp_-dbHcGlihqc_wG
      gho9fLMK8JOArYLcMDNQ",
  "header": {
    "alg": "RSA1_5",
    "kid": "frodo.baggins@hobbiton.example"
  }
}
```

Figure 206: Recipient #1 JSON

5.13.4. Encrypting the Key to the Second Recipient

The following are generated before encrypting the CEK for the second recipient:

- Ephemeral EC private key on the same curve as the EC public key; this example uses the private key from Figure 207.

```
{
  "kty": "EC",
  "crv": "P-384",
  "x": "Uzdvk3pi5wKCRc1izp5_r0OjeqT-I68i8g2b8mva8diRhsE2xAn2Dt
      MRb25Ma2CX",
  "y": "VDrRyFJh-Kwd1EjAgmj5Eo-CTHAZ53MC7PjjpLioy3ylEjI1pOMbw9
      1fzZ84pbfm",
  "d": "1DKHfTv-PiifVw2VBHM_ZiVcwOMxkOyANS_lQHJcrDxVY3jhVCvZPw
      MxJKIE793C"
}
```

Figure 207: Ephemeral public key for Recipient #2, in JWK format

Performing the "ECDH-ES+A256KW" key encryption operation over the CEK (Figure 202 with the following:

- Static Elliptic Curve public key (Figure 108).
- Ephemeral Elliptic Curve private key (Figure 207.

produces the following encrypted key:

```
ExInT0io9BqBMYF6-maw5tZlgoZXThD1zWKsHixJuw_elY4gSSId_w
```

Figure 208: Recipient #2 Encrypted Key, base64url-encoded

The following are generated after encrypting the CEK for the second recipient:

- Recipient JWE Unprotected Header from Figure 209.

```
{
  "alg": "ECDH-ES+A256KW",
  "kid": "peregrin.took@tuckborough.example",
  "epk": {
    "kty": "EC",
    "crv": "P-384",
    "x": "Uzdvk3pi5wKCRc1izp5_r0OjeqT-I68i8g2b8mva8diRhsE2xAn2
        DtMRb25Ma2CX",
    "y": "VDrRyFJh-Kwd1EjAgmj5Eo-CTHAZ53MC7PjjpLioy3ylEjI1pOMb
        w91fzZ84pbfm"
  }
}
```

Figure 209: Recipient #2 JWE Per-recipient Unprotected Header JSON

The following is the assembled second recipient JSON:

```
{
  "encrypted_key": "ExInT0io9BqBMYF6-maw5tZlgoZXThD1zWKsHixJuw
      _elY4gSSId_w",
  "header": {
    "alg": "ECDH-ES+A256KW",
    "kid": "peregrin.took@tuckborough.example",
    "epk": {
      "kty": "EC",
      "crv": "P-384",
      "x": "Uzdvk3pi5wKCRc1izp5_r0OjeqT-I68i8g2b8mva8diRhsE2xA
          n2DtMRb25Ma2CX",
      "y": "VDrRyFJh-Kwd1EjAgmj5Eo-CTHAZ53MC7PjjpLioy3ylEjI1pO
          Mbw91fzZ84pbfm"
    }
  }
}
```

Figure 210: Recipient #2 JSON

5.13.5. Encrypting the Key to the Third Recipient

The following are generated before encrypting the CEK for the third recipient:

- Initialization vector/nonce for key wrapping; this example uses the initialization vector/nonce from Figure 211

```
AvpeoPZ9Ncn9mkBn
```

Figure 211: Recipient #2 Initialization Vector, base64url-encoded

Performing the "A256GCMKW" key encryption operation over the CEK (Figure 202) with the following:

- AES symmetric key (Figure 138; and
- Initialization vector/nonce ((Figure 211

produces the following:

- Encrypted key from Figure 212.
- Key wrap authentication tag from Figure 213

```
a7CclAejo_7JSuPB8zeagxXRam8dwCfmkt9-WyTpS1E
```

Figure 212: Recipient #3 Encrypted Key, base64url-encoded

```
59Nqh1LlYtVIhfD3pgRGvw
```

Figure 213: Recipient #3 Authentication Tag, base64url-encoded

The following are generated after encrypting the CEK for the third recipient:

- Recipient JWE Unprotected Header; this example uses the header from Figure 214.

```
{
  "alg": "A256GCMKW",
  "kid": "18ec08e1-bfa9-4d95-b205-2b4dd1d4321d",
  "tag": "59Nqh1LlYtVIhfD3pgRGvw",
  "iv": "AvpeoPZ9Ncn9mkBn"
}
```

Figure 214: Recipient #3 JWE Per-recipient Unprotected Header JSON

The following is the assembled third recipient JSON:

```
{
  "encrypted_key": "a7CclAejo_7JSuPB8zeagxXRam8dwCfmkt9-WyTpS1
      E",
  "header": {
    "alg": "A256GCMKW",
    "kid": "18ec08e1-bfa9-4d95-b205-2b4dd1d4321d",
    "tag": "59Nqh1LlYtVIhfD3pgRGvw",
    "iv": "AvpeoPZ9Ncn9mkBn"
  }
}
```

Figure 215: Recipient #3 JSON

5.13.6. Encrypting the Content

The following are generated before encrypting the content:

- JWE Protected Header; this example uses the header from Figure 216, encoded to [RFC4648] base64url as Figure 217.

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

Figure 216: JWE Protected Header JSON

```
eyJlbmMiOiJBMTI4Q0JDLUhTMjU2In0
```

Figure 217: JWE Protected Header, base64url-encoded

Performing the content encryption operation over the Plaintext (Figure 72) with the following:

- CEK (Figure 202),
- Initialization vector/nonce (Figure 203), and
- JWE Protected Header (Figure 217) as the authenticated data

produces the following:

- Ciphertext from Figure 218
- Authentication tag from Figure 219

```
ajm2Q-OpPXCr7-MHXicknb1lsxLdXxK_yLds0KuhJzfWK04SjdxQeSw2L9mu3a
_k1C55kCQ_3xlkcVKC5yr__Is48VOoK0k63_QRM9tBURMFqLByJ8vOYQX0oJW4
VUHJLmGhF-tVQWB7Kz8mr8zeE7txF0MSaP6ga7-siYxStR7_G07Thd1jh-zGT0
wxM5g-VRORtq0K6AXpLlwEqRp7pkt2zRM0ZAXqSpe1O6FJ7FHLDyEFnD-zDIZu
kLpCbzhzMDLLw2-8I14FQrgi-iEuzHgIJFIJn2wh9Tj0cg_kOZy9BqMRZbmYXM
Y9YQjorZ_P_JYG3ARAIF3OjDNqpdYe-K_5Q5crGJSDNyij_ygEiItR5jssQVH2
ofDQdLChtazE
```

Figure 218: Ciphertext, base64url-encoded

```
BESYyFN7T09KY7i8zKs5_g
```

Figure 219: Authentication Tag, base64url-encoded

The following is generated after encrypting the plaintext:

- JWE Shared Unprotected Header parameters; this example uses the header from Figure 220.

```
{
  "cty": "text/plain"
}
```

Figure 220: JWE Shared Unprotected Header JSON

5.13.7. Output Results

The following compose the resulting JWE object:

- Recipient #1 JSON (Figure 206)
- Recipient #2 JSON (Figure 210)
- Recipient #3 JSON (Figure 215)
- Initialization vector/nonce (Figure 203)
- Ciphertext (Figure 218)
- Authentication tag (Figure 219)

The Compact Serialization is not presented because it does not support this use case; the JSON Flattened Serialization is not presented because there is more than one recipient.

The resulting JWE object using the JSON General Serialization:

```
{
```

```
     "recipients": [
       {
         "encrypted_key": "dYOD28kab0Vvf4ODgxVAJXgHcSZICSOp8M51zj
             wj4w6Y5G4XJQsNNIBiqyvUUAOcpL7S7-cFe7Pio7gV_Q06WmCSa-
             vhW6me4bWrBf7cHwEQJdXihidAYWVajJIaKMXMvFRMV6iDlRr076
             DFthg2_AV0_tSiV6xSEIFqt1xnYPpmP91tc5WJDOGb-wqjw0-b-S
             1laS11QVbuP78dQ7Fa0zAVzzjHX-xvyM2wxj_otxr9clN1LnZMbe
             YSrRicJK5xodvWgkpIdkMHo4LvdhRRvzoKzlic89jFWPlnBq_V4n
             5trGuExtp_-dbHcGlihqc_wGgho9fLMK8JOArYLcMDNQ",
         "header": {
           "alg": "RSA1_5",
           "kid": "frodo.baggins@hobbiton.example"
         }
       },
       {
         "encrypted_key": "ExInT0io9BqBMYF6-maw5tZlgoZXThD1zWKsHi
             xJuw_elY4gSSId_w",
         "header": {
           "alg": "ECDH-ES+A256KW",
           "kid": "peregrin.took@tuckborough.example",
           "epk": {
             "kty": "EC",
             "crv": "P-384",
             "x": "Uzdvk3pi5wKCRc1izp5_r0OjeqT-I68i8g2b8mva8diRhs
                 E2xAn2DtMRb25Ma2CX",
             "y": "VDrRyFJh-Kwd1EjAgmj5Eo-CTHAZ53MC7PjjpLioy3ylEj
                 I1pOMbw91fzZ84pbfm"
           }
         }
       },
       {
         "encrypted_key": "a7CclAejo_7JSuPB8zeagxXRam8dwCfmkt9-Wy
             TpS1E",
         "header": {
           "alg": "A256GCMKW",
           "kid": "18ec08e1-bfa9-4d95-b205-2b4dd1d4321d",
           "tag": "59Nqh1LlYtVIhfD3pgRGvw",
           "iv": "AvpeoPZ9Ncn9mkBn"
         }
       }
     ],
     "unprotected": {
       "cty": "text/plain"
     },
     "protected": "eyJlbmMiOiJBMTI4Q0JDLUhTMjU2In0",
     "iv": "VgEIHY20EnzUtZFl2RpB1g",
     "ciphertext": "ajm2Q-OpPXCr7-MHXicknb1lsxLdXxK_yLds0KuhJzfWK
         04SjdxQeSw2L9mu3a_k1C55kCQ_3xlkcVKC5yr__Is48VOoK0k63_QRM
```

```
        9tBURMFqLByJ8vOYQX0oJW4VUHJLmGhF-tVQWB7Kz8mr8zeE7txF0MSa
        P6ga7-siYxStR7_G07Thd1jh-zGT0wxM5g-VRORtq0K6AXpLlwEqRp7p
        kt2zRM0ZAXqSpe1O6FJ7FHLDyEFnD-zDIZukLpCbzhzMDLLw2-8I14FQ
        rgi-iEuzHgIJFIJn2wh9Tj0cg_kOZy9BqMRZbmYXMY9YQjorZ_P_JYG3
        ARAIF3OjDNqpdYe-K_5Q5crGJSDNyij_ygEiItR5jssQVH2ofDQdLCht
        azE",
    "tag": "BESYyFN7T09KY7i8zKs5_g"
  }
```

Figure 221: JSON General Serialization

6. Nesting Signatures and Encryption

This example illustrates nesting a JSON Web Signature (JWS) structure within a JSON Web Encryption (JWE) structure. The signature uses the "PS256" (RSASSA-PSS) algorithm; the encryption uses the "RSA-OAEP" (RSAES-OAEP) key encryption algorithm and the "A128GCM" (AES-GCM) content encryption algorithm.

Note that RSASSA-PSS uses random data to generate the signature, and RSAES-OAEP uses random data to generate the ciphertext; it might not be possible to exactly replicate the results in this section.

Note that whitespace is added for readability as described in Section 1.1.

6.1. Signing Input Factors

The following are supplied before beginning the signing operation:

- Payload content; this example uses the JSON Web Token (JWT) [I-D.ietf-oauth-json-web-token] content from Figure 222, encoded as [RFC4648] base64url to produce Figure 223.
- RSA private key; this example uses the key from Figure 224.
- "alg" parameter of "PS256".

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

Figure 222: Payload content, in JSON format

```
eyJpc3MiOiJob2JiaXRvbi5leGFtcGxlIiwiZXhwIjoxMzAwODE5MzgwLCJodH
RwOi8vZXhhbXBsZS5jb20vaXNfcm9vdCI6dHJ1ZX0
```

Figure 223: Payload content, base64url-encoded

```
{
  "kty": "RSA",
  "kid": "hobbiton.example",
  "use": "sig",
  "n": "kNrPIBDXMU6fcyv5i-QHQAQ-K8gsC3HJb7FYhYaw8hXbNJa-t8q0lD
      KwLZgQXYV-ffWxXJv5GGrlZE4GU52lfMEegTDzYTrRQ3tepgKFjMGg6I
      y6fkl1ZNsx2gEonsnlShfzA9GJwRTmtKPbk1s-hwx1IU5AT-AIelNqBg
      cF2vE5W25_SGGBoaROVdUYxqETDggM1z5cKV4ZjDZ8-lh4oVB07bkac6
      LQdHpJUUySH_Er20DXx30Kyi97PciXKTS-QKXnmm8ivyRCmux22ZoPUi
      nd2BKC5OiG4MwALhaL2Z2k8CsRdfy-7dg7z41Rp6D0ZeEvtaUp4bX4aK
      raL4rTfw",
  "e": "AQAB",
  "d": "ZLe_TIxpE9-W_n2VBa-HWvuYPtjvxwVXClJFOpJsdea8g9RMx34qEO
      EtnoYc2un3CZ3LtJi-mju5RAT8YSc76YJds3ZVw0UiO8mMBeG6-iOnvg
      obobNx7K57-xjTJZU72EjOr9kB7z6ZKwDDq7HFyCDhUEcYcHFVc7iL_6
      TibVhAhOFONWlqlJgEgwVYd0rybNGKifdnpEbwyHoMwY6HM1qvnEFgP7
      iZ0YzHUT535x6jj4VKcdA7ZduFkhUauysySEW7mxZM6fj1vdjJIy9LD1
      fIz30Xv4ckoqhKF5GONU6tNmMmNgAD6gIViyEle1PrIxl1tBhCI14bRW
      -zrpHgAQ",
  "p": "yKWYoNIAqwMRQlgIBOdT1NIcbDNUUs2Rh-pBaxD_mIkweMt4Mg-0-B
      2iSYvMrs8horhonV7vxCQagcBAATGW-hAafUehWjxWSH-3KccRM8toL4
      e0q7M-idRDOBXSoe7Z2-CV2x_ZCY3RP8qp642R13WgXqGDIM4MbUkZSj
      cY9-c",
  "q": "uND4o15V30KDzf8vFJw589p1vlQVQ3NEilrinRUPHkkxaAzDzccGgr
      WMWpGxGFFnNL3w5CqPLeU76-5IVYQq0HwYVl0hVXQHr7sgaGu-483Ad3
      ENcL23FrOnF45m7_2ooAstJDe49MeLTTQKrSIBl_SKvqpYvfSPTczPcZ
      kh9Kk",
  "dp": "jmTnEoq2qqa8ouaymjhJSCnsveUXnMQC2gAneQJRQkFqQu-zV2PKP
      KNbPvKVyiF5b2-L3tM3OW2d2iNDyRUWXlT7V5l0KwPTABSTOnTqAmYCh
      Gi8kXXdlhcrtSvXldBakC6saxwI_TzGGY2MVXzc2ZnCvCXHV4qjSxOrf
      P3pHFU",
  "dq": "R9FUvU88OVzEkTkXl3-5-WusE4DjHmndeZIlu3rifBdfLpq_P-iWP
      BbGaq9wzQ1c-J7SzCdJqkEJDv5yd2C7rnZ6kpzwBh_nmL8zscAk1qsun
      nt9CJGAYz7-sGWy1JGShFazfP52ThB4rlCJ0YuEaQMrIzpY77_oLAhpm
      DA0hLk",
  "qi": "S8tC7ZknW6hPITkjcwttQOPLVmRfwirRlFAViuDb8NW9CrV_7F2Oq
      UZCqmzHTYAumwGFHI1WVRep7anleWaJjxC_1b3fq_al4qH3Pe-EKiHg6
      IMazuRtZLUROcThrExDbF5dYbsciDnfRUWLErZ4N1Be0bnxYuPqxwKd9
      QZwMo0"
}
```

Figure 224: RSA 2048-bit Private Key, in JWK format

6.2. Signing Operation

The following are generated to complete the signing operation:

- JWS Protected Header; this example uses header from Figure 225, encoded using [RFC4648] base64url to produce Figure 226.

```
{
  "alg": "PS256",
  "typ": "JWT"
}
```

Figure 225: JWS Protected Header JSON

```
eyJhbGciOiJQUzI1NiIsInR5cCI6IkpXVCJ9
```

Figure 226: JWS Protected Header, base64url-encoded

Performing the signature operation over the combined JWS Protected Header (Figure 226) and Payload content (Figure 222) produces the following signature:

```
dPpMqwRZxFYi1UfcDAaf8M99o7kwUWtiXZ-ByvVuJih4MhJ_aZqciprz0OWaIA
kIvn1qskChirjKvY9ESZNUCP4JjvfyPS-nqjJxYoA5ztWOyFk2cZNIPXjcJXSQ
wXPO9tEe-v4VSqgD0aKHqPxYog4N6Cz1lKph1U1sYDSI67_bLL7elg_vkjfMp5
_W5l5LuUYGMeh6hxQIaIUXf9EwV2JmvTMuZ-vBOWy0Sniy1EFo72CRTvmtrIf5
AROo5MNliY3KtUxeP-SOmD-LEYwW9SlkohYzMVAZDDOrVbv7KVRHpeYNaK75KE
QqdCEEkS_rskZS-Qtt_nlegTWh1mEYaA
```

Figure 227: JWS Signature, base64url-encoded

6.3. Signing Output

The following compose the resulting JWS object:

- JWS Protected Header (Figure 226))
- Payload content (Figure 223)
- Signature (Figure 227)

The resulting JWS object using the Compact Serialization (which is the plaintext input to the proceeding encryption operation):

```
eyJhbGciOiJQUzI1NiIsInR5cCI6IkpXVCJ9
.
eyJpc3MiOiJob2JiaXRvbi5leGFtcGxlIiwiZXhwIjoxMzAwODE5MzgwLCJodH
RwOi8vZXhhbXBsZS5jb20vaXNfcm9vdCI6dHJ1ZX0
.
dPpMqwRZxFYi1UfcDAaf8M99o7kwUWtiXZ-ByvVuJih4MhJ_aZqciprz0OWaIA
kIvn1qskChirjKvY9ESZNUCP4JjvfyPS-nqjJxYoA5ztWOyFk2cZNIPXjcJXSQ
wXPO9tEe-v4VSqgD0aKHqPxYog4N6Cz1lKph1U1sYDSI67_bLL7elg_vkjfMp5
_W5l5LuUYGMeh6hxQIaIUXf9EwV2JmvTMuZ-vBOWy0Sniy1EFo72CRTvmtrIf5
AROo5MNliY3KtUxeP-SOmD-LEYwW9SlkohYzMVAZDDOrVbv7KVRHpeYNaK75KE
QqdCEEkS_rskZS-Qtt_nlegTWh1mEYaA
```

Figure 228: Compact Serialization

6.4. Encryption Input Factors

The following are supplied before beginning the encryption process:

- Plaintext content; this example uses the content from Figure 228.
- RSA public key; this example use the key from Figure 84.
- "alg" parameter of "RSA-OAEP".
- "enc" parameter of "A128GCM".

6.5. Encryption Generated Factors

The following are generated before encrypting:

- AES symmetric key as the Content Encryption Key (CEK); this example uses the key from Figure 229.
- Initialization vector/nonce; this example uses the initialization vector/nonce from Figure 230.

```
0RHSNYwN-6-2QBGsYTZLSQ
```

Figure 229: Content Encryption Key, base64url-encoded

```
GbX1i9kXz0sxXPmA
```

Figure 230: Initialization vector, base64url-encoded

6.6. Encrypting the Key

Performing the key encryption operation over the CEK (Figure 229) with the RSA key (Figure 84) produces the following encrypted key:

```
a0JHRoITfpX4qRewImjlStn8m3CPxBV1ueYlVhjurCyrBg3I7YhCRYjphDOOS4
E7rXbr2Fn6NyQq-A-gqT0FXqNjVOGrG-bi13mwy7RoYhjTkBEC6P7sMYMXXx4g
zMedpiJHQVeyI-zkZV7A9matpgevAJWrXzOUysYGTtwoSN6gtUVtlLaivjvb21
O0ul4YxSHV-ByK1kyeetRp_fuYJxHoKLQL9P424sKx2WGYb4zsBIPF4ssl_e5I
R7nany-25_UmC2urosNkoFz9cQ82MypZP8gqbQJyPN-Fpp4Z-5o6yV64x6yzDU
F_5JCIdl-Qv6H5dMVIY7q1eKpXcV1lWO_2FefEBqXxXvIjLeZivjNkzogCq3-I
apSjVFnMjBxjpYLT8muaawo1yy1XXMuinIpNcOY3n4KKrXLrCcteX85m4IIHMZ
a38s1Hpr56fPPseMA-Jltmt-a9iEDtOzhtxz8AXy9tsCAZV2XBWNG8c3kJusAa
mBKOYwfk7JhLRDgOnJjlJLhn7TI4UxDp9dCmUXEN6z0v23W15qJIEXNJtqnblp
ymooeWAHCT4e_Owbim1g0AEpTHUdA2iiLNs9WTX_H_TXuPC8yDDhi1smxS_X_x
pkIHkiIHWDOLx03BpqDTivpKkBYwqP2UZkcxqX2Fo_GnVrNwlK7Lgxw6FSQvDO
0
```

Figure 231: Encrypted Key, base64url-encoded

6.7. Encrypting the Content

The following are generated before encrypting the plaintext:

- JWE Protected Header; this example uses the the header from Figure 232, encoded using [RFC4648] base64url to produce Figure 233.

```
{
  "alg": "RSA-OAEP",
  "cty": "JWT",
  "enc": "A128GCM"
}
```

Figure 232: JWE Protected Header JSON

```
eyJhbGciOiJSU0EtT0FFUCIsImN0eSI6IkpXVCIsImVuYyI6IkExMjhHQ00ifQ
```

Figure 233: JWE Protected Header, base64url-encoded

Performing the content encryption operation over the Plaintext (Figure 228) with the following:

- CEK (Figure 229);
- Initialization vector/nonce (Figure 230); and
- JWE Protected Header (Figure 233) as authenticated data.

produces the following:

- Ciphertext from Figure 234.
- Authentication tag from Figure 235.

```
SZI4IvKHmwpazl_pJQXX3mHv1ANnOU4Wf9-utWYUcKrBNgCe2OFMf66cSJ8k2Q
kxaQD3_R60MGE9ofomwtky3GFxMeGRjtpMt9OAvVLsAXB0_UTCBGyBg3C2bWLX
qZlfJAAoJRUPRk-BimYZY81zVBuIhc7HsQePCpu33SzMsFHjn4lP_idrJz_glZ
TNgKDt8zdnUPauKTKDNOH1DD4fuzvDYfDIAfqGPyL5sVRwbiXpXdGokEszM-9C
hMPqW1QNhzuX_Zul3bvrJwr7nuGZs4cUScY3n8yE3AHCLurgls-A9mz1X38xEa
ulV18l4Fg9tLejdkAuQZjPbqeHQBJe4IwGD5Ee0dQ-Mtz4NnhkIWx-YKBb_Xo2
zI3Q_1sYjKUuis7yWW-HTr_vqvFt0bj7WJf2vzB0TZ3dvsoGaTvPH2dyWwumUr
lx4gmPUzBdwTO6ubfYSDUEEz5py0d_OtWeUSYcCYBKD-aM7tXg26qJo21gYjLf
hn9zy-W19sOCZGuzgFjPhawXHpvnj_t-0_ES96kogjJLxS1IMU9Y5XmnwZMyNc
9EIwnogsCg-hVuvzyP0sIruktmI94_SL1xgMl7o03phcTMxtlMizR88NKU1WkB
siXMCjy1Noue7MD-ShDp5dmM
```

Figure 234: Ciphertext, base64url-encoded

```
KnIKEhN8U-3C9s4gtSpjSw
```

Figure 235: Authentication tag, base64url-encoded

6.8. Encryption Output

The following compose the resulting JWE object:

- JWE Protected Header (Figure 233)
- Encrypted key (Figure 231)
- Initialization vector/nonce (Figure 230)
- Ciphertext (Figure 234)
- Authentication Tag (Figure 235)

The resulting JWE object using the Compact serialization:

```
eyJhbGciOiJSU0EtT0FFUCIsImN0eSI6IkpXVCIsImVuYyI6IkExMjhHQ00ifQ
.
a0JHRoITfpX4qRewImjlStn8m3CPxBV1ueYlVhjurCyrBg3I7YhCRYjphDOOS4
E7rXbr2Fn6NyQq-A-gqT0FXqNjVOGrG-bi13mwy7RoYhjTkBEC6P7sMYMXXx4g
zMedpiJHQVeyI-zkZV7A9matpgevAJWrXzOUysYGTtwoSN6gtUVtlLaivjvb21
O0ul4YxSHV-ByK1kyeetRp_fuYJxHoKLQL9P424sKx2WGYb4zsBIPF4ssl_e5I
R7nany-25_UmC2urosNkoFz9cQ82MypZP8gqbQJyPN-Fpp4Z-5o6yV64x6yzDU
F_5JCIdl-Qv6H5dMVIY7q1eKpXcV1lWO_2FefEBqXxXvIjLeZivjNkzogCq3-I
apSjVFnMjBxjpYLT8muaawo1yy1XXMuinIpNcOY3n4KKrXLrCcteX85m4IIHMZ
a38s1Hpr56fPPseMA-Jltmt-a9iEDtOzhtxz8AXy9tsCAZV2XBWNG8c3kJusAa
mBKOYwfk7JhLRDgOnJjlJLhn7TI4UxDp9dCmUXEN6z0v23W15qJIEXNJtqnblp
ymooeWAHCT4e_Owbim1g0AEpTHUdA2iiLNs9WTX_H_TXuPC8yDDhi1smxS_X_x
pkIHkiIHWDOLx03BpqDTivpKkBYwqP2UZkcxqX2Fo_GnVrNwlK7Lgxw6FSQvDO
0
.
GbX1i9kXz0sxXPmA
.
SZI4IvKHmwpazl_pJQXX3mHv1ANnOU4Wf9-utWYUcKrBNgCe2OFMf66cSJ8k2Q
kxaQD3_R60MGE9ofomwtky3GFxMeGRjtpMt9OAvVLsAXB0_UTCBGyBg3C2bWLX
qZlfJAAoJRUPRk-BimYZY81zVBuIhc7HsQePCpu33SzMsFHjn4lP_idrJz_glZ
TNgKDt8zdnUPauKTKDNOH1DD4fuzvDYfDIAfqGPyL5sVRwbiXpXdGokEszM-9C
hMPqW1QNhzuX_Zul3bvrJwr7nuGZs4cUScY3n8yE3AHCLurgls-A9mz1X38xEa
ulV18l4Fg9tLejdkAuQZjPbqeHQBJe4IwGD5Ee0dQ-Mtz4NnhkIWx-YKBb_Xo2
zI3Q_1sYjKUuis7yWW-HTr_vqvFt0bj7WJf2vzB0TZ3dvsoGaTvPH2dyWwumUr
lx4gmPUzBdwTO6ubfYSDUEEz5py0d_OtWeUSYcCYBKD-aM7tXg26qJo21gYjLf
hn9zy-W19sOCZGuzgFjPhawXHpvnj_t-0_ES96kogjJLxS1IMU9Y5XmnwZMyNc
9EIwnogsCg-hVuvzyP0sIruktmI94_SL1xgMl7o03phcTMxtlMizR88NKU1WkB
siXMCjy1Noue7MD-ShDp5dmM
.
KnIKEhN8U-3C9s4gtSpjSw
```

Figure 236: Compact Serialization

The resulting JWE object using the JSON General Serialization:

```
{
  "recipients": [
    {
      "encrypted_key": "a0JHRoITfpX4qRewImjlStn8m3CPxBV1ueYlVh
          jurCyrBg3I7YhCRYjphDOOS4E7rXbr2Fn6NyQq-A-gqT0FXqNjVO
          GrG-bi13mwy7RoYhjTkBEC6P7sMYMXXx4gzMedpiJHQVeyI-zkZV
          7A9matpgevAJWrXzOUysYGTtwoSN6gtUVtlLaivjvb21O0ul4YxS
          HV-ByK1kyeetRp_fuYJxHoKLQL9P424sKx2WGYb4zsBIPF4ssl_e
          5IR7nany-25_UmC2urosNkoFz9cQ82MypZP8gqbQJyPN-Fpp4Z-5
          o6yV64x6yzDUF_5JCIdl-Qv6H5dMVIY7q1eKpXcV1lWO_2FefEBq
          XxXvIjLeZivjNkzogCq3-IapSjVFnMjBxjpYLT8muaawo1yy1XXM
          uinIpNcOY3n4KKrXLrCcteX85m4IIHMZa38s1Hpr56fPPseMA-Jl
          tmt-a9iEDtOzhtxz8AXy9tsCAZV2XBWNG8c3kJusAamBKOYwfk7J
          hLRDgOnJjlJLhn7TI4UxDp9dCmUXEN6z0v23W15qJIEXNJtqnblp
          ymooeWAHCT4e_Owbim1g0AEpTHUdA2iiLNs9WTX_H_TXuPC8yDDh
          i1smxS_X_xpkIHkiIHWDOLx03BpqDTivpKkBYwqP2UZkcxqX2Fo_
          GnVrNwlK7Lgxw6FSQvDO0"
    }
  ],
  "protected": "eyJhbGciOiJSU0EtT0FFUCIsImN0eSI6IkpXVCIsImVuYy
      I6IkExMjhHQ00ifQ",
  "iv": "GbX1i9kXz0sxXPmA",
  "ciphertext": "SZI4IvKHmwpazl_pJQXX3mHv1ANnOU4Wf9-utWYUcKrBN
      gCe2OFMf66cSJ8k2QkxaQD3_R60MGE9ofomwtky3GFxMeGRjtpMt9OAv
      VLsAXB0_UTCBGyBg3C2bWLXqZlfJAAoJRUPRk-BimYZY81zVBuIhc7Hs
      QePCpu33SzMsFHjn4lP_idrJz_glZTNgKDt8zdnUPauKTKDNOH1DD4fu
      zvDYfDIAfqGPyL5sVRwbiXpXdGokEszM-9ChMPqW1QNhzuX_Zul3bvrJ
      wr7nuGZs4cUScY3n8yE3AHCLurgls-A9mz1X38xEaulV18l4Fg9tLejd
      kAuQZjPbqeHQBJe4IwGD5Ee0dQ-Mtz4NnhkIWx-YKBb_Xo2zI3Q_1sYj
      KUuis7yWW-HTr_vqvFt0bj7WJf2vzB0TZ3dvsoGaTvPH2dyWwumUrlx4
      gmPUzBdwTO6ubfYSDUEEz5py0d_OtWeUSYcCYBKD-aM7tXg26qJo21gY
      jLfhn9zy-W19sOCZGuzgFjPhawXHpvnj_t-0_ES96kogjJLxS1IMU9Y5
      XmnwZMyNc9EIwnogsCg-hVuvzyP0sIruktmI94_SL1xgMl7o03phcTMx
      tlMizR88NKU1WkBsiXMCjy1Noue7MD-ShDp5dmM",
  "tag": "KnIKEhN8U-3C9s4gtSpjSw"
}
```


Figure 237: JSON General Serialization

The resulting JWE object using the JSON Flattened Serialization:

```
{
  "encrypted_key": "a0JHRoITfpX4qRewImjlStn8m3CPxBV1ueYlVhjurC
      yrBg3I7YhCRYjphDOOS4E7rXbr2Fn6NyQq-A-gqT0FXqNjVOGrG-bi13
      mwy7RoYhjTkBEC6P7sMYMXXx4gzMedpiJHQVeyI-zkZV7A9matpgevAJ
      WrXzOUysYGTtwoSN6gtUVtlLaivjvb21O0ul4YxSHV-ByK1kyeetRp_f
      uYJxHoKLQL9P424sKx2WGYb4zsBIPF4ssl_e5IR7nany-25_UmC2uros
      NkoFz9cQ82MypZP8gqbQJyPN-Fpp4Z-5o6yV64x6yzDUF_5JCIdl-Qv6
      H5dMVIY7q1eKpXcV1lWO_2FefEBqXxXvIjLeZivjNkzogCq3-IapSjVF
      nMjBxjpYLT8muaawo1yy1XXMuinIpNcOY3n4KKrXLrCcteX85m4IIHMZ
      a38s1Hpr56fPPseMA-Jltmt-a9iEDtOzhtxz8AXy9tsCAZV2XBWNG8c3
      kJusAamBKOYwfk7JhLRDgOnJjlJLhn7TI4UxDp9dCmUXEN6z0v23W15q
      JIEXNJtqnblpymooeWAHCT4e_Owbim1g0AEpTHUdA2iiLNs9WTX_H_TX
      uPC8yDDhi1smxS_X_xpkIHkiIHWDOLx03BpqDTivpKkBYwqP2UZkcxqX
      2Fo_GnVrNwlK7Lgxw6FSQvDO0",
  "protected": "eyJhbGciOiJSU0EtT0FFUCIsImN0eSI6IkpXVCIsImVuYy
      I6IkExMjhHQ00ifQ",
  "iv": "GbX1i9kXz0sxXPmA",
  "ciphertext": "SZI4IvKHmwpazl_pJQXX3mHv1ANnOU4Wf9-utWYUcKrBN
      gCe2OFMf66cSJ8k2QkxaQD3_R60MGE9ofomwtky3GFxMeGRjtpMt9OAv
      VLsAXB0_UTCBGyBg3C2bWLXqZlfJAAoJRUPRk-BimYZY81zVBuIhc7Hs
      QePCpu33SzMsFHjn4lP_idrJz_glZTNgKDt8zdnUPauKTKDNOH1DD4fu
      zvDYfDIAfqGPyL5sVRwbiXpXdGokEszM-9ChMPqW1QNhzuX_Zul3bvrJ
      wr7nuGZs4cUScY3n8yE3AHCLurgls-A9mz1X38xEaulV18l4Fg9tLejd
      kAuQZjPbqeHQBJe4IwGD5Ee0dQ-Mtz4NnhkIWx-YKBb_Xo2zI3Q_1sYj
      KUuis7yWW-HTr_vqvFt0bj7WJf2vzB0TZ3dvsoGaTvPH2dyWwumUrlx4
      gmPUzBdwTO6ubfYSDUEEz5py0d_OtWeUSYcCYBKD-aM7tXg26qJo21gY
      jLfhn9zy-W19sOCZGuzgFjPhawXHpvnj_t-0_ES96kogjJLxS1IMU9Y5
      XmnwZMyNc9EIwnogsCg-hVuvzyP0sIruktmI94_SL1xgMl7o03phcTMx
      tlMizR88NKU1WkBsiXMCjy1Noue7MD-ShDp5dmM",
  "tag": "KnIKEhN8U-3C9s4gtSpjSw"
}
```


Figure 238: JSON Flattened Serialization

7. Security Considerations

This document is designed to provide examples for developers to use in checking their implementations. As such it does not follow some of the security considerations and recommendations in the core documents. For instance:

- it does not always generate a new CEK value for every encrypted example;
- it does not always generate a new IV value for every encrypted example; and

- it does not always generate a new ephemeral key for every ephemeral key example.

For each example, data that is expected to be generated for each signing or encryption operation is isolated to sections titled "Generated Factors".

8. IANA Considerations

This document has no actions for IANA.

9. References

9.1. Normative References

[I-D.ietf-jose-json-web-algorithms]
Jones, M., "JSON Web Algorithms (JWA)", draft-ietf-jose-json-web-algorithms-38 (work in progress), December 2014.

[I-D.ietf-jose-json-web-encryption]
Jones, M. and J. Hildebrand, "JSON Web Encryption (JWE)", draft-ietf-jose-json-web-encryption-38 (work in progress), December 2014.

[I-D.ietf-jose-json-web-key]
Jones, M., "JSON Web Key (JWK)", draft-ietf-jose-json-web-key-38 (work in progress), December 2014.

[I-D.ietf-jose-json-web-signature]
Jones, M., Bradley, J., and N. Sakimura, "JSON Web Signature (JWS)", draft-ietf-jose-json-web-signature-38 (work in progress), December 2014.

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

9.2. Informative References

[I-D.ietf-oauth-json-web-token]
Jones, M., Bradley, J., and N. Sakimura, "JSON Web Token (JWT)", draft-ietf-oauth-json-web-token-32 (work in progress), December 2014.

[LOTR-FELLOWSHIP]
Tolkien, J. and C. Tolkien, "The Fellowship of the Ring", ISBN 9780061917702, March 2009.

[RFC1951] Deutsch, P., "DEFLATE Compressed Data Format Specification version 1.3", RFC 1951, May 1996.

[RFC7095] Kewisch, P., "jCard: The JSON Format for vCard", RFC 7095, January 2014.

Appendix A. Acknowledgements

Most of the examples herein use quotes and character names found in the novel "The Fellowship of the Ring" [LOTR-FELLOWSHIP], written by J. R. R. Tolkien.

Thanks to Richard Barnes, Brian Campbell, Mike Jones, and Jim Schaad for input and review of text. Thanks to Brian Campbell for verifying Compact Serialization examples.

Author’s Address

Matthew Miller
Cisco Systems, Inc.

Email: mamille2@cisco.com

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.

Table of Contents

1. Introduction

The JSON Web Algorithms (JWA) specification registers cryptographic algorithms and identifiers to be used with the JSON Web Signature (JWS) [JWS], JSON Web Encryption (JWE) [JWE], and JSON Web Key (JWK) [JWK] specifications. It defines several IANA registries for these identifiers. All these specifications utilize JavaScript Object Notation (JSON) [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 be 32 octets long. The octet sequence representations MUST NOT be shortened to omit any leading zero octets contained in the values.

3. Concatenate the two octet sequences in the order R and then S. (Note that many ECDSA implementations will directly produce this concatenation as their output.)

4. The resulting 64 octet sequence is the JWS Signature value.

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

alg 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.mcgrew-aead-aes-cbc-hmac-sha2], performing the same cryptographic computations, but with the Initialization Vector and Authentication Tag values remaining separate, rather than being concatenated with the Ciphertext value in the output representation. This option is discussed in Appendix B of that specification. This algorithm family is a generalization of the algorithm family in [I-D.mcgrew-aead-aes-cbc-hmac-sha2], and can be used to implement those algorithms.

5.2.1. Conventions Used in Defining AES_CBC_HMAC_SHA2

We use the following notational conventions.

CBC-PKCS5-ENC(X, P) denotes the AES CBC encryption of P using PKCS #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:

- "crv"
- "x"

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

- "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:

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

These values are registered in the IANA JSON Web Key Elliptic Curve registry defined in Section 7.6. Additional "crv" values 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 ceiling(log-base-2(n)/8) octets (where n is the order of the curve).

6.3. Parameters for RSA Keys

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

- Protocol Category: JSON Object Signing and Encryption (JOSE)
- Registry Location: http://www.iana.org/assignments/jose
- Webpage Title: (same as the protocol category)
- 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

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

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

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

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

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

- Algorithm Analysis Documents(s): n/a

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

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

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

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

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

- Algorithm Name: "PS384"
- Algorithm Description: RSASSA-PSS using SHA-384 and MGF1 with SHA-384

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

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

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

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

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

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

- Algorithm Analysis Documents(s): n/a

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

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

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

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

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

- Algorithm Name: "ECDH-ES+A128KW"
- Algorithm Description: ECDH-ES using Concat KDF and "A128KW" wrapping
- Algorithm Usage Location(s): "alg"

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

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

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

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

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

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

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

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

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

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

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

- Algorithm Name: "A256CBC-HS512"

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

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

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

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

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

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

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

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

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

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

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

7.3. JSON Web Encryption Compression Algorithms Registry

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

7.3.1. Registration Template

Compression Algorithm Value:
The name requested (e.g., "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

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

7.4. JSON Web Key Types Registry

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

The implementation requirements of a key type may be changed over time 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.

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

- "kty" Parameter Value: "RSA"

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

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

7.5. JSON Web Key Parameters Registration

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

7.5.1. Registry Contents

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

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

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

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

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

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

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

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

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

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

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

- Change Controller: IESG
- Specification Document(s): Section 6.3.2.5 of [[this document]]

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

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

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

7.6. JSON Web Key Elliptic Curve Registry

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

The implementation requirements of a curve may be changed over time 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

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

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

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

8. Security Considerations

All of the security issues 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-saslprepbis] to prepare a password supplied directly by a user before performing key derivation and encryption.

10. References

10.1. Normative References

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

[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-saslprepbis-13 (work in progress), December 2014.

[I-D.mcgrew-aead-aes-cbc-hmac-sha2]
McGrew, D., Foley, J., and K. Paterson, "Authenticated Encryption with AES-CBC and HMAC-SHA", draft-mcgrew-aead-aes-cbc-hmac-sha2-05 (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	1.2.840.113549.2.9
HS384	http://www.w3.org/2001/04/xmldsig-more#hmac-sha384	HmacSHA384	1.2.840.113549.2.10
HS512	http://www.w3.org/2001/04/xmldsig-more#hmac-sha512	HmacSHA512	1.2.840.113549.2.11
RS256	http://www.w3.org/2001/04/xmldsig-more#rsa-sha256	SHA256withRSA	1.2.840.113549.1.1.11
RS384	http://www.w3.org/2001/04/xmldsig-more#rsa-sha384	SHA384withRSA	1.2.840.113549.1.1.12
RS512	http://www.w3.org/2001/04/xmldsig-more#rsa-sha512	SHA512withRSA	1.2.840.113549.1.1.13
ES256	http://www.w3.org/2001/04/xmldsig-more#ecdsa-sha256	SHA256withECDSA	1.2.840.10045.4.3.2
ES384	http://www.w3.org/2001/04/xmldsig-more#ecdsa-sha384	SHA384withECDSA	1.2.840.10045.4.3.3
ES512	http://www.w3.org/2001/04/xmldsig-more#ecdsa-sha512	SHA512withECDSA	1.2.840.10045.4.3.4
PS256	http://www.w3.org/2007/05/xmldsig-more#sha256-rsa-MGF1	SHA256withRSAandMGF1	1.2.840.113549.1.1.10
PS384	http://www.w3.org/2007/05/xmldsig-more#sha384-rsa-MGF1	SHA384withRSAandMGF1	1.2.840.113549.1.1.10
PS512	http://www.w3.org/2007/05/xmldsig-more#sha512-rsa-MGF1	SHA512withRSAandMGF1	1.2.840.113549.1.1.10

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 & http://www.w3.org/2009/xmlenc11#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/NoPadding	2.16.840.1.101.3.4.1.26
A256GCM	http://www.w3.org/2009/xmlenc11#aes256-gcm	AES/GCM/NoPadding	2.16.840.1.101.3.4.1.46

Appendix B. Test Cases for AES_CBC_HMAC_SHA2 Algorithms

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

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

B.1. Test Cases for AES_128_CBC_HMAC_SHA_256

AES_128_CBC_HMAC_SHA_256

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

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

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

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

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

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

AL =      00 00 00 00 00 00 01 50

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

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

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

B.2. Test Cases for AES_192_CBC_HMAC_SHA_384

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

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

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

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

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

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

AL =      00 00 00 00 00 00 01 50

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

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

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

B.3. Test Cases for AES_256_CBC_HMAC_SHA_512

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

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

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

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

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

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

AL =      00 00 00 00 00 00 01 50

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

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

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

Appendix C. Example ECDH-ES Key Agreement Computation

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

In this example, a 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":"weNJy2HscCSM6AEDTDg04biOvhFhyyWvOHQfeF_PxMQ",
 "y":"e8lnCO-AlStT-NJVX-crhB7QRYhiix03illJOVAOyck",
 "d":"VEmDZpDXXK8p8N0Cndsxs924q6nS1RXFASRl6BfUqdw"
}
```

Header Parameter values used in this example are as follows. In this example, the "apu" (agreement PartyUInfo) parameter value is the base64url encoding of the UTF-8 string "Alice" and the "apv" (agreement PartyVInfo) parameter value is the base64url encoding of the UTF-8 string "Bob". The "epk" parameter is used to communicate the 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_xSffzlPWrHEVI30DHM_4egVwt3NQqeUD7nMFpps"
  }
}
```

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

Z
: This is set to the ECDH-ES key agreement output. (This value is often not directly exposed by libraries, due to NIST security requirements, and only serves as an input to a KDF.) In this example, Z is 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.mcgrew-aead-aes-cbc-hmac-sha2] specification, upon which the AES_CBC_HMAC_SHA2 algorithms are based, was written by David A. McGrew and Kenny Paterson. The test cases for AES_CBC_HMAC_SHA2 are based upon those for [I-D.mcgrew-aead-aes-cbc-hmac-sha2] by John Foley.

Matt Miller wrote Using JavaScript Object Notation (JSON) Web Encryption (JWE) for Protecting JSON Web Key (JWK) Objects

[I-D.miller-jose-jwe-protected-jwk], which the password-based encryption content of this draft is based upon.

This specification is the work of the JOSE Working Group, which includes dozens of active and dedicated participants. In particular, the following individuals contributed ideas, feedback, and wording that influenced this specification:

Dirk Balfanz, Richard Barnes, 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

- Clarified the definitions of UTF8(STRING) and ASCII(STRING).

-39

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

-38

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

-37

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

-36

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

-35

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

-34

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

-33

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

-32

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

-31

- Referenced NIST SP 800-57 for guidance on key lifetimes.

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

-30

- Cleaned up the reference syntax in a few places.

- Applied minor wording changes to the Security Considerations section.

-29

- 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

- 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.)

- 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

- Described additional security considerations.

- Updated the JCA and XMLENC parameters for "RSA-OAEP-256" and the JCA parameters for "A128KW", "A192KW", "A256KW", and "ECDH-ES".

-26

- Added algorithm identifier "RSA-OAEP-256" for RSAES OAEP using SHA-256 and MGF1 with SHA-256.

- Clarified that the ECDSA signature values R and S are represented as octet sequences as defined in Section 2.3.7 of SEC1 [SEC1].

- Noted that octet sequences are depicted using JSON array notation.

- Updated references, including to W3C specifications.

-25

- Corrected an external section number reference that had changed.

-24

- 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.

- Updated the JSON reference to RFC 7159.

-23

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

-22

- Corrected RFC 2119 terminology usage.

- Replaced references to draft-ietf-json-rfc4627bis with RFC 7158.

-21

- 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.

- Changed some references from being normative to informative, addressing issue #90.

-20

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

-19

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

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

-18

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

- Added and used Description registry fields.

-17

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

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

- Replaced verbose repetitive phases such as "base64url encode the octets of the UTF-8 representation of X" with mathematical notation such as "BASE64URL(UTF8(X))".

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

- Changes to address minor issue #53.

-16

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

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

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

-15

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

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

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

- Updated the example ECDH-ES key agreement values.

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

-14

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

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

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

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

- Added an example ECDH-ES key agreement computation.

-13

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

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

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

-12

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

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

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

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

-11

- Removed the Encrypted Key value from the AAD computation since it is already effectively integrity protected by the encryption process. The AAD value now only contains the representation of the JWE Encrypted Header.

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

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

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

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

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

-10

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

-09

- Expanded the scope of the JWK parameters to include private and symmetric key representations, as specified by draft-jones-jose-json-private-and-symmetric-key-00.

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

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

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

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

- Replaced "A128CBC+HS256" and "A256CBC+HS512" with "A128CBC-HS256" and "A256CBC-HS512". The new algorithms perform the same cryptographic computations as [I-D.mcgrew-aead-aes-cbc-hmac-sha2], but with the Initialization Vector and Authentication Tag values remaining separate from the Ciphertext value in the output representation. Also deleted the header parameters "epu" (encryption PartyUInfo) and "epv" (encryption PartyVInfo), since they are no longer used.

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

-08

- Changed the name of the JWK key type parameter from "alg" to "kty".

- Replaced uses of the term "AEAD" with "Authenticated Encryption", since the term AEAD in the RFC 5116 sense implied the use of a particular data representation, rather than just referring to the class of algorithms that perform authenticated encryption with associated data.

- Applied editorial improvements suggested by Jeff Hodges. Many of these simplified the terminology used.

- Added seriesInfo information to Internet Draft references.

-07

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

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

- Made several local editorial changes to clean up loose ends left over from to the decision to only support block encryption methods providing integrity.

-06

- Removed the "int" and "kdf" parameters and defined the new composite Authenticated Encryption algorithms "A128CBC+HS256" and "A256CBC+HS512" to replace the former uses of AES CBC, which required the use of separate integrity and key derivation functions.

- Included additional values in the Concat KDF calculation -- the desired output size and the algorithm value, and optionally PartyUInfo and PartyVInfo values. Added the optional header parameters "apu" (agreement PartyUInfo), "apv" (agreement PartyVInfo), "epu" (encryption PartyUInfo), and "epv" (encryption PartyVInfo).

- Changed the name of the JWK RSA exponent parameter from "exp" to "xpo" so as to allow the potential use of the name "exp" for a future extension that might define an expiration parameter for keys. (The "exp" name is already used for this purpose in the JWT specification.)

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

-05

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

- Updated open issues.

-04

- Added text requiring that any leading zero bytes be retained in base64url encoded key value representations for fixed-length values.

- Added this language to Registration Templates: "This name is case sensitive. Names that match other registered names in a case insensitive manner SHOULD NOT be accepted."

- Described additional open issues.

- Applied editorial suggestions.

-03

- Always use a 128 bit "authentication tag" size for AES GCM, regardless of the key size.

- Specified that use of a 128 bit IV is REQUIRED with AES CBC. It was previously RECOMMENDED.

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

-02

- For AES GCM, use the "additional authenticated data" parameter to provide integrity for the header, encrypted key, and ciphertext and use the resulting "authentication tag" value as the JWE Authentication Tag.
- Defined minimum required key sizes for algorithms without specified key sizes.

- Defined KDF output key sizes.
- Specified the use of PKCS #5 padding with AES CBC.
- Generalized text to allow key agreement to be employed as an alternative to key wrapping or key encryption.
- Clarified that ECDH-ES is a key agreement algorithm.
- Required implementation of AES-128-KW and AES-256-KW.
- Removed the use of "A128GCM" and "A256GCM" for key wrapping.
- Removed "A512KW" since it turns out that it’s not a standard algorithm.
- Clarified the relationship between "typ" header parameter values and MIME types.
- Generalized language to refer to Message Authentication Codes (MACs) rather than Hash-based Message Authentication Codes (HMACs) unless in a context specific to HMAC algorithms.
- Established registries: JSON Web Signature and Encryption Header Parameters, JSON Web Signature and Encryption Algorithms, JSON Web Signature and Encryption "typ" Values, JSON Web Key Parameters, and JSON Web Key Algorithm Families.
- Moved algorithm-specific definitions from JWK to JWA.
- Reformatted to give each member definition its own section heading.

-01

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

- Corrected the Magic Signatures reference.
- Made other editorial improvements suggested by JOSE working group participants.

-00

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

Author's Address

Michael B. Jones
Microsoft

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

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

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 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):

- 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 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:

- "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 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.

- 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 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

- 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 [[this document]]

- 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 [[this document]]

- 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 [[this document]]

- 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 [[this document]]

- 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 [[this document]]

- 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 [[this document]]

- 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 [[this document]]

- 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 [[this document]]

- 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 [[this document]]

- 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 [[this document]]

- 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 [[this document]]

- 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 [[this document]]

- 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 [[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.mcgrew-aead-aes-cbc-hmac-sha2] McGrew, D., Foley, J., and K. Paterson, "Authenticated Encryption with AES-CBC and HMAC-SHA", draft-mcgrew-aead-aes-cbc-hmac-sha2-05 (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:

- 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 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
      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 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
      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 Content Encryption Key (CEK). In this example, the value is:

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

A.3.3. Key Encryption

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

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

The resulting JWE Encrypted Key value is:

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

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

```
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 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_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 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

- Clarified the definitions of UTF8(STRING) and ASCII(STRING).

-39

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

-38

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

-37

- Restricted algorithm names to using only ASCII characters.

- 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.

- Added the CRT parameter values to example RSA private key representations.

-36

- Defined a flattened JWE JSON Serialization syntax, which is optimized for the single recipient case.

- Clarified where white space and line breaks may occur in JSON objects by referencing Section 2 of RFC 7159.

-35

- Addressed AppsDir reviews by Ray Polk.

-34

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

-33

- Noted that certificate thumbprints are also sometimes known as certificate fingerprints.

- Changed to use the term "authenticated encryption" instead of "encryption", where appropriate.

- Acknowledged additional contributors.

-32

- Addressed Gen-ART review comments by Russ Housley.

- Addressed secdir review comments by Scott Kelly, Tero Kivinen, and Stephen Kent.

-31

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

-30

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

-29

- 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

- 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.)
- Revised the introduction to the Security Considerations section. Also moved a security consideration item here from the JWA draft.

-27

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

-26

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

-25

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

-24

- Corrected complete JSON Serialization example.

- 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.

- Updated the JSON reference to RFC 7159.

- Thanked Eric Rescorla for helping to author of most of the drafts of this specification and removed him from the current author list.

-23

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

-22

- Corrected RFC 2119 terminology usage.

- Replaced references to draft-ietf-json-rfc4627bis with RFC 7158.

-21

- Changed some references from being normative to informative, addressing issue #90.

- Applied review comments to the JSON Serialization section, addressing issue #178.

-20

- Made terminology definitions more consistent, addressing issue #165.

- Restructured the JSON Serialization section to call out the parameters used in hanging lists, addressing issue #178.

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

-19

- Reordered the key selection parameters.

-18

- 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.

- Changes to address editorial and minor issues #89, #135, #165, #174, #175, #177, #179, and #180.

- Used Header Parameter Description registry field.

-17

- Refined the "typ" and "cty" definitions to always be MIME Media Types, with the omission of "application/" prefixes recommended for brevity, addressing issue #50.

- 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.

- 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.

- Replaced verbose repetitive phases such as "base64url encode the octets of the UTF-8 representation of X" with mathematical notation such as "BASE64URL(UTF8(X))".

- Header Parameters and processing rules occurring in both JWS and JWE are now referenced in JWS by JWE, rather than duplicated, addressing issue #57.

- 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

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

-15

- Clarified that it is an application decision which recipients' encrypted content must successfully validate for the JWE to be accepted, addressing issue #35.

- Changes to address editorial issues #34, #164, and #169.

-14

- 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

- 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

- Clarified that the "typ" and "cty" header parameters are used in an application-specific manner and have no effect upon the JWE processing.

- Replaced the MIME types "application/jwe+json" and "application/jwe" with "application/jose+json" and "application/jose".

- 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.

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

- 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.

- Restored use of the term "AEAD".

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

-11

- Added Key Identification section.

- 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.

- 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.)

- Shortened the names "authentication_tag" to "tag" and "initialization_vector" to "iv" in the JWE JSON Serialization, addressing issue #20.

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

- Removed suggested compact serialization for multiple recipients.

- Changed the MIME type name "application/jwe-js" to "application/jwe+json", addressing issue #22.

- Tightened the description of the "crit" (critical) header parameter.

-10

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

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

-09

- Added JWE JSON Serialization, as specified by draft-jones-jose-jwe-json-serialization-04.

- Registered "application/jwe-js" MIME type and "JWE-JS" typ header parameter value.

- 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.

- Corrected "x5c" description. This addressed issue #12.
- Changed from using the term "byte" to "octet" when referring to 8 bit values.
- Added Key Management Mode definitions to terminology section and used the defined terms to provide clearer key management instructions. This addressed issue #5.
- Added text about preventing the recipient from behaving as an oracle during decryption, especially when using RSAES-PKCS1-V1_5.
- Changed from using the term "Integrity Value" to "Authentication Tag".
- Changed member name from "integrity_value" to "authentication_tag" in the JWE JSON Serialization.
- 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.
- Replaced "A128CBC+HS256" and "A256CBC+HS512" with "A128CBC-HS256" and "A256CBC-HS512". The new algorithms perform the same cryptographic computations as [I-D.mcgrew-aead-aes-cbc-hmac-sha2], but with the Initialization Vector and Authentication Tag values remaining separate from the Ciphertext value in the output representation. Also deleted the header parameters "epu" (encryption PartyUInfo) and "epv" (encryption PartyVInfo), since they are no longer used.

-08

- Replaced uses of the term "AEAD" with "Authenticated Encryption", since the term AEAD in the RFC 5116 sense implied the use of a particular data representation, rather than just referring to the class of algorithms that perform authenticated encryption with associated data.
- Applied editorial improvements suggested by Jeff Hodges and Hannes Tschofenig. Many of these simplified the terminology used.
- Clarified statements of the form "This header parameter is OPTIONAL" to "Use of this header parameter is OPTIONAL".

- Added a Header Parameter Usage Location(s) field to the IANA JSON Web Signature and Encryption Header Parameters registry.

- Added seriesInfo information to Internet Draft references.

-07

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

- Updated values for example AES CBC calculations.

- Made several local editorial changes to clean up loose ends left over from to 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

- Removed the "int" and "kdf" parameters and defined the new composite Authenticated Encryption algorithms "A128CBC+HS256" and "A256CBC+HS512" to replace the former uses of AES CBC, which required the use of separate integrity and key derivation functions.

- Included additional values in the Concat KDF calculation -- the desired output size and the algorithm value, and optionally PartyUInfo and PartyVInfo values. Added the optional header parameters "apu" (agreement PartyUInfo), "apv" (agreement PartyVInfo), "epu" (encryption PartyUInfo), and "epv" (encryption PartyVInfo). Updated the KDF examples accordingly.

- 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.

- 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.

- Added an AES Key Wrap example.

- Reordered the encryption steps so CMK creation is first, when required.

- Correct statements in examples about which algorithms produce reproducible results.

-05

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

-04

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

-03

- 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.
- 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.
- 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.

- Moved description of how to determine whether a header is for a JWS or a JWE from the JWT spec to the JWE spec.
- Added complete encryption examples for both Authenticated Encryption and non-Authenticated Encryption algorithms.
- Added complete key derivation examples.
- Added "Collision Resistant Namespace" to the terminology section.
- Reference ITU.X690.1994 for DER encoding.
- Added Registry Contents sections to populate registry values.
- Numerous editorial improvements.

-02

- 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.
- Defined KDF output key sizes.
- Generalized text to allow key agreement to be employed as an alternative to key wrapping or key encryption.
- Changed compression algorithm from gzip to DEFLATE.
- Clarified that it is an error when a "kid" value is included and no matching key is found.
- Clarified that JWEs with duplicate Header Parameter Names MUST be rejected.
- Clarified the relationship between "typ" header parameter values and MIME types.
- Registered application/jwe MIME type and "JWE" typ header parameter value.
- 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).

- Added suggestion on defining additional header parameters such as "x5t#S256" in the future for certificate thumbprints using hash algorithms other than SHA-1.

- Specify RFC 2818 server identity validation, rather than RFC 6125 (paralleling the same decision in the OAuth specs).

- 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.

- Reformatted to give each header parameter its own section heading.

-01

- Added an integrity check for non-Authenticated Encryption algorithms.

- Added "jpk" and "x5c" header parameters for including JWK public keys and X.509 certificate chains directly in the header.

- Clarified that this specification is defining the JWE Compact Serialization. Referenced the new JWE-JS spec, which defines the JWE JSON Serialization.

- Added text "New header parameters should be introduced sparingly since an implementation that does not understand a parameter MUST reject the JWE".

- 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.

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

-00

- Created the initial IETF draft based upon draft-jones-json-web-encryption-02 with no normative changes.

- 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

Table of Contents

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:

- "sig" (signature)
- "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:

- "sign" (compute digital signature or MAC)
- "verify" (verify digital signature or MAC)
- "encrypt" (encrypt content)
- "decrypt" (decrypt content and validate decryption, if applicable)
- "wrapKey" (encrypt key)
- "unwrapKey" (decrypt key and validate decryption, if applicable)
- "deriveKey" (derive key)
- "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.

- Protocol Category: JSON Object Signing and Encryption (JOSE)
- Registry Location: http://www.iana.org/assignments/jose
- Webpage Title: (same as the protocol category)
- 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

- Parameter Name: "kty"
- Parameter Description: Key Type
- Used with "kty" Value(s): *
- Parameter Information Class: Public
- Change Controller: IESG
- Specification Document(s): Section 4.1 of [[this document]]

- Parameter Name: "use"
- Parameter Description: Public Key Use
- Used with "kty" Value(s): *
- Parameter Information Class: Public
- Change Controller: IESG
- Specification Document(s): Section 4.2 of [[this document]]

- Parameter Name: "key_ops"
- Parameter Description: Key Operations
- Used with "kty" Value(s): *
- Parameter Information Class: Public
- Change Controller: IESG
- Specification Document(s): Section 4.3 of [[this document]]

- Parameter Name: "alg"
- Parameter Description: Algorithm
- Used with "kty" Value(s): *

- Parameter Information Class: Public
- Change Controller: IESG
- Specification Document(s): Section 4.4 of [[this document]]

- Parameter Name: "kid"
- Parameter Description: Key ID
- Used with "kty" Value(s): *
- Parameter Information Class: Public
- Change Controller: IESG
- Specification Document(s): Section 4.5 of [[this document]]

- Parameter Name: "x5u"
- Parameter Description: X.509 URL
- Used with "kty" Value(s): *
- Parameter Information Class: Public
- Change Controller: IESG
- Specification Document(s): Section 4.6 of [[this document]]

- Parameter Name: "x5c"
- Parameter Description: X.509 Certificate Chain
- Used with "kty" Value(s): *
- Parameter Information Class: Public
- Change Controller: IESG
- Specification Document(s): Section 4.7 of [[this document]]

- Parameter Name: "x5t"
- Parameter Description: X.509 Certificate SHA-1 Thumbprint
- Used with "kty" Value(s): *
- Parameter Information Class: Public
- Change Controller: IESG
- Specification Document(s): Section 4.8 of [[this document]]

- Parameter Name: "x5t#S256"
- Parameter Description: X.509 Certificate SHA-256 Thumbprint
- Used with "kty" Value(s): *
- Parameter Information Class: Public
- Change Controller: IESG
- 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

- Use Member Value: "sig"
- Use Description: Digital Signature or MAC
- Change Controller: IESG
- Specification Document(s): Section 4.2 of [[this document]]

- Use Member Value: "enc"
- Use Description: Encryption
- Change Controller: IESG
- 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

- Key Operation Value: "sign"
- Key Operation Description: Compute digital signature or MAC
- Change Controller: IESG
- Specification Document(s): Section 4.3 of [[this document]]

- Key Operation Value: "verify"
- Key Operation Description: Verify digital signature or MAC
- Change Controller: IESG
- Specification Document(s): Section 4.3 of [[this document]]

- Key Operation Value: "encrypt"
- Key Operation Description: Encrypt content
- Change Controller: IESG
- Specification Document(s): Section 4.3 of [[this document]]

- Key Operation Value: "decrypt"
- Key Operation Description: Decrypt content and validate decryption, if applicable

- Change Controller: IESG
- Specification Document(s): Section 4.3 of [[this document]]

- Key Operation Value: "wrapKey"
- Key Operation Description: Encrypt key
- Change Controller: IESG
- Specification Document(s): Section 4.3 of [[this document]]

- Key Operation Value: "unwrapKey"
- Key Operation Description: Decrypt key and validate decryption, if applicable
- Change Controller: IESG
- Specification Document(s): Section 4.3 of [[this document]]

- Key Operation Value: "deriveKey"
- Key Operation Description: Derive key
- Change Controller: IESG
- Specification Document(s): Section 4.3 of [[this document]]

- Key Operation Value: "deriveBits"
- Key Operation Description: Derive bits not to be used as a key
- Change Controller: IESG
- 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

- Parameter Name: "keys"
- Parameter Description: Array of JWK values
- Change Controller: IESG
- 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.MediaTypes] 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.

- Type Name: application
- Subtype Name: jwk+json
- Required Parameters: n/a
- Optional Parameters: n/a
- Encoding considerations: 8bit; application/jwk+json values are represented as JSON object; UTF-8 encoding SHOULD be employed for the JSON object.
- Security Considerations: See the Security Considerations section of [[this document]]
- Interoperability Considerations: n/a
- Published Specification: [[this document]]
- Applications that use this media type: OpenID Connect, Salesforce, Google, Android, Windows Azure, W3C WebCrypto API, numerous others
- Fragment identifier considerations: n/a
- Additional Information: Magic number(s): n/a, File extension(s): n/a, Macintosh file type code(s): n/a
- Person & email address to contact for further information: Michael B. Jones, mbj@microsoft.com

- Intended Usage: COMMON
- Restrictions on Usage: none
- Author: Michael B. Jones, mbj@microsoft.com
- Change Controller: IESG
- Provisional registration? No

- Type Name: application
- Subtype Name: jwk-set+json
- Required Parameters: n/a
- Optional Parameters: n/a
- Encoding considerations: 8bit; application/jwk-set+json values are represented as a JSON Object; UTF-8 encoding SHOULD be employed for the JSON object.
- Security Considerations: See the Security Considerations section of [[this document]]
- Interoperability Considerations: n/a
- Published Specification: [[this document]]
- Applications that use this media type: OpenID Connect, Salesforce, Google, Android, Windows Azure, W3C WebCrypto API, numerous others
- Fragment identifier considerations: n/a
- Additional Information: Magic number(s): n/a, File extension(s): n/a, Macintosh file type code(s): n/a
- Person & email address to contact for further information: Michael B. Jones, mbj@microsoft.com
- Intended Usage: COMMON
- Restrictions on Usage: none
- Author: Michael B. Jones, mbj@microsoft.com
- Change Controller: IESG
- 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-xmldsig-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.MediaTypes] 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 Diffe-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-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/>.

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":"MKBCTNIcKUSDii11ySs3526iDZ8AiTo7Tu6KPAqv7D4",
     "y":"4Etl6SRW2YiLUrN5vfvVHuhp7x8PxltmWWlbbM4IFyM",
     "use":"enc",
     "kid":"1"},

    {"kty":"RSA",
     "n": "0vx7agoebGcQSuuPiLJXZptN9nndrQmbXEps2aiAFbWhM78LhWx
4cbbfAAtVT86zwu1RK7aPFFxuhDR1L6tSoc_BJECPebWKRXjBZCiFV4n3oknjhMs
tn64tZ_2W-5JsGY4Hc5n9yBXArwl93lqt7_RN5w6Cf0h4QyQ5v-65YGjQR0_FDW2
QvzqY368QQMicAtaSqzs8KJZgnYb9c7d0zgdAZHzu6qMQvRL5hajrn1n91CbOpbI
SD08qNLyrdkt-bFTWhAI4vMQFh6WeZu0fM4lFd2NcRwr3XPksINHaQ-G_xBniIqb
w0Ls1jF44-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":"MKBCTNIcKUSDii11ySs3526iDZ8AiTo7Tu6KPAqv7D4",
     "y":"4Etl6SRW2YiLUrN5vfvVHuhp7x8PxltmWWlbbM4IFyM",
     "d":"870MB6gfuTJ4HtUnUvYMyJpr5eUZNP4Bk43bVdj3eAE",
     "use":"enc",
     "kid":"1"},

    {"kty":"RSA",
     "n":"0vx7agoebGcQSuuPiLJXZptN9nndrQmbXEps2aiAFbWhM78LhWx4
cbbfAAtVT86zwu1RK7aPFFxuhDR1L6tSoc_BJECPebWKRXjBZCiFV4n3oknjhMst
n64tZ_2W-5JsGY4Hc5n9yBXArwl93lqt7_RN5w6Cf0h4QyQ5v-65YGjQR0_FDW2Q
vzqY368QQMicAtaSqzs8KJZgnYb9c7d0zgdAZHzu6qMQvRL5hajrn1n91CbOpbIS
D08qNLyrdkt-bFTWhAI4vMQFh6WeZu0fM4lFd2NcRwr3XPksINHaQ-G_xBniIqbw
0Ls1jF44-csFCur-kEgU8awapJzKnqDKgw",
     "e":"AQAB",
     "d":"X4cTteJY_gn4FYPsXB8rdXix5vwsg1FLN5E3EaG6RJoVH-HLLKD9
M7dx5oo7GURknchnrRweUkC7hT5fJLM0WbFAKNLWY2vv7B6NqXSzUvxT0_YSfqij
wp3RTzlBaCxWp4doFk5N2o8Gy_nHNKroADIkJ46pRUohsXywbReAdYaMwFs9tv8d
_cPVY3i07a3t8MN6TNwm0dSawm9v47UiCl3Sk5ZiG7xojPLu4sbg1U2jx4IBTNBz
nbJSzFHK66jT8bgkuqsk0GjskDJk19Z4qwjwbsnn4j2WBii3RL-Us2lGVkY8fkFz
me1z0HbIkfz0Y6mqnOYtqc0X4jfcKoAC8Q",
     "p":"83i-7IvMGXoMXCskv73TKr8637FiO7Z27zv8oj6pbWUQyLPQBQxtPV
nwD20R-60eTDmD2ujnMt5PoqMrm8RfmNhVWDtjjMmCMjOpSXicFHj7XOuVIYQyqV
WlWEh6dN36GVZYk93N8Bc9vY41xy8B9RzzOGVQzXvNEvn7O0nVbfs",
     "q":"3dfOR9cuYq-0S-mkFLzgItgMEfFzB2q3hWehMuG0oCuqnb3vobLyum
qjVZQO1dIrdwgTnCdpYzBcOfW5r370AFXjiWft_NGEiovonizhKpo9VVS78TzFgx
kIdrecRezsZ-1kYd_s1qDbxtkDEgfAITAG9LUnADun4vIcb6yelxk",
     "dp":"G4sPXkc6Ya9y8oJW9_ILj4xuppu0lzi_H7VTkS8xj5SdX3coE0oim
YwxIi2emTAue0UOa5dpgFGyBJ4c8tQ2VF402XRugKDTP8akYhFo5tAA77Qe_Nmtu
YZc3C3m3I24G2GvR5sSDxUyAN2zq8Lfn9EUms6rY3Ob8YeiKkTiBj0",
     "dq":"s9lAH9fggBsoFR8Oac2R_E2gw282rT2kGOAhvIllETE1efrA6huUU
vMfBcMpn8lqeW6vzznYY5SSQF7pMdC_agI3nG8Ibp1BUb0JUiraRNqUfLhcQb_d9
GF4Dh7e74WbRsobRonujTYN1xCaP6TO61jvWrX-L18txXw494Q_cgk",
     "qi":"GyM_p6JrXySiz1toFgKbWV-JdI3jQ4ypu9rbMWx3rQJBfmt0FoYzg
UIZEVFEcOqwemRN81zoDAaa-Bk0KWNGDjJHZDdDmFhW3AN7lI-puxk_mHZGJ11rx
yR8O55XLSe3SPmRfKwZI6yU24ZxvQKFYItdldUKGzO6Ia6zTKhAVRU",
     "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":"GawgguFyGrWKav7AX4VKUg"},

    {"kty":"oct",
     "k":"AyM1SysPpbyDfgZld3umj1qzKObwVMkoqQ-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_PdiVGKPrqzmDIsLI7sA25VEnHU1uCLNwBuUiCO11_-7dYbsr4iJmG0Q
 u2j8DsVyT1azpJC_NG84Ty5KKthuCaPod7iI7w0LK9orSMhBEwwZDCxTWq4a
 YWAchc8t-emd9qOvWtVMDC2BXksRngh6X5bUYLy6AyHKvj-nUy1wgzjYQDwH
 MTplCoLtU-o-8SNnZ1tmRoGE9uJkBLdh5gFENabWnU5m1ZqZPdwS-qo-meMv
 VfJb6jJVWRpl2SUtCnYG2C32qvbWbjZ_jBPD5eunqsIo1vQ",
 "e":"AQAB",
 "x5c":
  ["MIIDQjCCAiqgAwIBAgIGATz/FuLiMA0GCSqGSIb3DQEBBQUAMGIxCzAJB
  gNVBAYTAlVTMQswCQYDVQQIEwJDTzEPMA0GA1UEBxMGRGVudmVyMRwwGgYD
  VQQKExNQaW5nIElkZW50aXR5IENvcnAuMRcwFQYDVQQDEw5CcmlhbiBDYW1
  wYmVsbDAeFw0xMzAyMjEyMzI5MTVaFw0xODA4MTQyMjI5MTVaMGIxCzAJBg
  NVBAYTAlVTMQswCQYDVQQIEwJDTzEPMA0GA1UEBxMGRGVudmVyMRwwGgYDV
  QQKExNQaW5nIElkZW50aXR5IENvcnAuMRcwFQYDVQQDEw5CcmlhbiBDYW1w
  YmVsbDCCASIwDQYJKoZIhvcNAQEBBQADggEPADCCAQoCggEBAL64zn8/QnH
  YMeZ0LncoXaEde1fiLm1jHjmQsF/449IYALM9if6amFtPDy2yvz3YlRij66
  s5gyLCyO7ANuVRJx1NbgizcAblIgjtdf/u3WG7K+IiZhtELto/A7Fck9Ws6
  SQvzRvOE8uSirYbgmj6He4iO8NCyvaK0jIQRMMGQwsU1quGmFgHIXPLfnpn
  fajr1rVTAwtgV5LEZ4Iel+W1GC8ugMhyr4/p1MtcIM42EA8BzE6ZQqC7VPq
  PvEjZ2dbZkaBhPbiZAS3YeYBRDWm1p1OZtWamT3cEvqqPpnjL1XyW+oyVVk
  aZdklLQp2Btgt9qr21m42f4wTw+Xrp6rCKNb0CAwEAATANBgkqhkiG9w0BA
  QUFAAOCAQEAh8zGlfSlcI0o3rYDPBB07aXNswb4ECNIKG0CETTUxmXl9KUL
  +9gGlqCz5iWLOgWsnrcKcY0vXPG9J1r9AqBNTqNgHq2G03X09266X5CpOe1
  zFo+Owb1zxtp3PehFdfQJ610CDLEaS9V9Rqp17hCyybEpOGVwe8fnk+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":"t6Q8PWSi1dkJj9hTP8hNYFlvadM7DflW9mWepOJhJ66w7nyoK1gPNqFMSQRy
        O125Gp-TEkodhWr0iujjHVx7BcV0llS4w5ACGgPrcAd6ZcSR0-Iqom-QFcNP
        8Sjg086MwoqQU_LYywlAGZ21WSdS_PERyGFiNnj3QQlO8Yns5jCtLCRwLHL0
        Pb1fEv45AuRIuUfVcPySBWYnDyGxvjYGDSM-AqWS9zIQ2ZilgT-GqUmipg0X
        OC0Cc20rgLe2ymLHjpHciCKVAbY5-L32-lSeZO-Os6U15_aXrk9Gw8cPUaX1
        _I8sLGuSiVdt3C_Fn2PZ3Z8i744FPFGGcG1qs2Wz-Q",
   "e":"AQAB",
   "d":"GRtbIQmhOZtyszfgKdg4u_N-R_mZGU_9k7JQ_jn1DnfTuMdSNprTeaSTyWfS
        NkuaAwnOEbIQVy1IQbWVV25NY3ybc_IhUJtfri7bAXYEReWaCl3hdlPKXy9U
        vqPYGR0kIXTQRqns-dVJ7jahlI7LyckrpTmrM8dWBo4_PMaenNnPiQgO0xnu
        ToxutRZJfJvG4Ox4ka3GORQd9CsCZ2vsUDmsXOfUENOyMqADC6p1M3h33tsu
        rY15k9qMSpG9OX_IJAXmxzAh_tWiZOwk2K4yxH9tS3Lq1yX8C1EWmeRDkK2a
        hecG85-oLKQt5VEpWHKmjOi_gJSdSgqcN96X52esAQ",
   "p":"2rnSOV4hKSN8sS4CgcQHFbs08XboFDqKum3sc4h3GRxrTmQdl1ZK9uw-PIHf
        QP0FkxXVrx-WE-ZEbrqivH_2iCLUS7wAl6XvARt1KkIaUxPPSYB9yk31s0Q8
        UK96E3_OrADAYtAJs-M3JxCLfNgqh56HDnETTQhH3rCT5T3yJws",
   "q":"1u_RiFDP7LBYh3N4GXLT9OpSKYP0uQZyiaZwBtOCBNJgQxaj10RWjsZu0c6I
        edis4S7B_coSKB0Kj9PaPaBzg-IySRvvcQuPamQu66riMhjVtG6TlV8CLCYK
        rYl52ziqK0E_ym2QnkwsUX7eYTB7LbAHRK9GqocDE5B0f808I4s",
   "dp":"KkMTWqBUefVwZ2_Dbj1pPQqyHSHjj90L5x_MOzqYAJMcLMZtbUtwKqvVDq3
        tbEo3ZIcohbDtt6SbfmWzggabpQxNxuBpoOOf_a_HgMXK_lhqigI4y_kqS1w
        Y52IwjUn5rgRrJ-yYo1h41KR-vz2pYhEAeYrhttWtxVqLCRViD6c",
   "dq":"AvfS0-gRxvn0bwJoMSnFxYcK1WnuEjQFluMGfwGitQBWtfZ1Er7t1xDkbN9
        GQTB9yqpDoYaN06H7CFtrkxhJIBQaj6nkF5KKS3TQtQ5qCzkOkmxIe3KRbBy
        mXxkb5qwUpX5ELD5xFc6FeiafWYY63TmmEAu_lRFCOJ3xDea-ots",
   "qi":"lSQi-w9CpyUReMErP1RsBLk7wNtOvs5EQpPqmuMvqW57NBUczScEoPwmUqq
        abu9V0-Py4dQ57_bapoKRu1R90bvuFnU63SHWEFglZQvJDMeAvmj4sm-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:

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

```
{
 "alg":"PBES2-HS256+A128KW",
 "p2s":"2WCTcJZ1Rvd_CJuJripQ1w",
 "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):

```
eyJhbGciOiJQQkVTMi1IUzI1NitBMTI4S1ciLCJwMnMiOiIyV0NUY0paMVJ2ZF9DSn
VKcmlwUTF3IiwicDJjIjo0MDk2LCJlbmMiOiJBMTI4Q0JDLUhTMjU2IiwiY3R5Ijoi
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-TR1uIxR9p6hsRQh9Tk7BA

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:

Ye9j1qs22DmRSAddIh-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):

```
AwhB8lxrlKjFn02LGWEqg27H4Tg9fyZAbFv3p5ZicHpj64QyHC44qqlZ3JEmnZTgQo
wIqZJ13jbyHB8LgePiqUJ1hf6M2HPLgzw8L-mEeQ0jvDUTrE07NtOerBk8bwBQyZ6g
0kQ3DEOIglfYxV8-FJvNBYwbqN1Bck6d_i7OtjSHV-8DIrp-3JcRIe05YKy3Oi34Z_
GOiAc1EK21B11c_AE11PII_wvvtRiUiG8YofQXakWd1_O98Kap-UgmyWPfreUJ3lJP
nbD4Ve95owEfMGLOPflo2MnjaTDCwQokoJ_xplQ2vNPz8iguLcHBoKllyQFJL2mOWB
wqhBo9Oj-O800as5mmLsvQMTflIrIEbbTMzHMBZ8EFW9fWwwFu0DWQJGkMNhmBZQ-3
lvqTc-M6-gWA6D8PDhONfP2Oib2HGizwG1iEaX8GRyUpfLuljCLIe1DkGOewhKuKkZ
h04DKNM5Nbugf2atmU9OP0Ldx5peCUtRG1gMVl7Qup5ZXHTjgPDr5b2N731UooCGAU
qHdgGhg0JVJ_ObCTdjsH4CF1SJsdUhrXvYx3HJh2Xd7CwJRzU_3Y1GxYU6-s3GFPbi
rfqqEipJDBTHpcoCmyrwYjYHFgnlqBZRotRrS95g8F95bRXqsaDY7UgQGwBQBwy665
d0zpvTasvfXf_c0MWAl-neFaKOW_Px6g4EUDjG1GWSXV9cLStLw_0ovdApDIFLHYHe
PyagyHjouQUuGiq7BsYwYrwaF06tgB8hV8omLNfMEmDPJaZUzMuHw6tBDwGkzD-tS_
ub9hxrpJ4UsOWnt5rGUyoN2N_c1-TQlXxm5oto14MxnoAyBQBpwIEgSH3Y4ZhwKBhH
PjSo0cdwuNdYbGPpb-YUvF-2NZzODiQ1OvWQBRHSbPWYz_xbGkgD504LRtqRwCO7CC
_CyyURi1sEssPVsMJRX_U4LFEOc82TiDdqjKOjRUfKK5rqLi8nBE9soQ0DSaOoFQZi
GrBrqxDsNYiAYAmxxkos-i3nX4qtByVx85sCE5U_0MqG7COxZWMOPEFrDaepUV-cOy
rvoUIng8i8ljKBKxETY2BgPegKBYCxsAUcAkKamSCC9AiBxA0UOHyhTqtlvMksO7AE
hNC2-YzPyx1FkhMoS4LLe6E_pFsMlmjA6P1NSge9C5G5tETYXGAn6b1xZbHtmwrPSc
ro9LWhVmAaA7_bxYObnFUxgWtK4vzzQBjZJ36UTk4OTB-JvKWgfVWCFsaw5WCHj6Oo
4jpO7d2yN7WMfAj2hTEabz9wumQ0TMhBduZ-QON3pYObSy7TSC1vVme0NJrwF_cJRe
hKTFmdlXGVldPxZCplr7ZQqRQhF8JP-l4mEQVnCaWGn9ONHlemczGOS-A-wwtnmwjI
B1V_vgJRf4FdpV-4hUk4-QLpu3-1lWFxrtZKcggq3tWTduRo5_QebQbUUT_VSCgsFc
OmyWKoj56lbxthN19hq1XGWbLGfrrR6MWh23vk01zn8FVwi7uFwEnRYSafsnWLa1Z5
TpBj9GvAdl2H9NHwzpB5NqHpZNkQ3NMDj13Fn8fzO0JB83Etbm_tnFQfcb13X3bJ15
Cz-Ww1MGhvIpGGnMBT_ADp9xSIyAM9dQ1yeVXk-AIgWBUlN5uyWSGyCxp0cJwx7HxM
38z0UIeBu-MytL-eqndM7LxytsVzCbjOTSVRmhYEMIzUAnS1gs7uMQAGRdgRIElTJE
SGMjb_4bZq9s6Ve1LKkSi0_QDsrABaLe55UY0zF4ZSfOV5PMyPtocwV_dcNPlxLgNA
D1BFX_Z9kAdMZQW6fAmsfFle0zAoMe4l9pMESH0JB4sJGdCKtQXj1cXNydDYozF7l8
H00BV_Er7zd6VtIw0MxwkFCTatsv_R-GsBCH218RgVPsfYhwVuT8R4HarpzsDBufC4
r8_c8fc9Z278sQ081jFjOja6L2x0N_ImzFNXU6xwO-Ska-QeuvYZ3X_L31ZOX4Llp-
7QSfgDoHnOxFv1Xws-D5mDHD3zxOup2b2TppdKTZb9eW2vxUVviM8OI9atBfPKMGAO
v9omA-6vv5IxUH0-lWMiHLQ_g8vnswp-Jav0c4t6URVUzujNOoNd_CBGGVnHiJTCHl
88LQxsqLHHIu4Fz-U2SGnlxGTj0-ihit2ELGRv4vO8E1BosTmf0cx3qgG0Pq0eOLBD
IHsrdZ_CCAiTc0HVkMbyq1M6qEhM-q5P6y1QCIrwg
```

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:


```
eyJhbGciOiJQQkVTMi1IUzI1NitBMTI4S1ciLCJwMnMiOiIyV0NUY0paMVJ2ZF9DSn
VKcmlwUTF3IiwicDJjIjo0MDk2LCJlbmMiOiJBMTI4Q0JDLUhTMjU2IiwiY3R5Ijoi
andrK2pzb24ifQ.
TrqXOwuNUfDV9VPTNbyGvEJ9JMjefAVn-TR1uIxR9p6hsRQh9Tk7BA.
Ye9j1qs22DmRSAddIh-VnA.
AwhB8lxrlKjFn02LGWEqg27H4Tg9fyZAbFv3p5ZicHpj64QyHC44qqlZ3JEmnZTgQo
wIqZJ13jbyHB8LgePiqUJ1hf6M2HPLgzw8L-mEeQ0jvDUTrE07NtOerBk8bwBQyZ6g
0kQ3DEOIglfYxV8-FJvNBYwbqN1Bck6d_i7OtjSHV-8DIrp-3JcRIe05YKy3Oi34Z_
GOiAc1EK21B11c_AE11PII_wvvtRiUiG8YofQXakWd1_O98Kap-UgmyWPfreUJ3lJP
nbD4Ve95owEfMGLOPflo2MnjaTDCwQokoJ_xplQ2vNPz8iguLcHBoKllyQFJL2mOWB
wqhBo9Oj-O800as5mmLsvQMTflIrIEbbTMzHMBZ8EFW9fWwwFu0DWQJGkMNhmBZQ-3
lvqTc-M6-gWA6D8PDhONfP2Oib2HGizwG1iEaX8GRyUpfLuljCLIe1DkGOewhKuKkZ
h04DKNM5Nbugf2atmU9OP0Ldx5peCUtRG1gMVl7Qup5ZXHTjgPDr5b2N731UooCGAU
qHdgGhg0JVJ_ObCTdjsH4CF1SJsdUhrXvYx3HJh2Xd7CwJRzU_3Y1GxYU6-s3GFPbi
rfqqEipJDBTHpcoCmyrwYjYHFgnlqBZRotRrS95g8F95bRXqsaDY7UgQGwBQBwy665
d0zpvTasvfXf_c0MWAl-neFaKOW_Px6g4EUDjG1GWSXV9cLStLw_0ovdApDIFLHYHe
PyagyHjouQUuGiq7BsYwYrwaF06tgB8hV8omLNfMEmDPJaZUzMuHw6tBDwGkzD-tS_
ub9hxrpJ4UsOWnt5rGUyoN2N_c1-TQlXxm5oto14MxnoAyBQBpwIEgSH3Y4ZhwKBhH
PjSo0cdwuNdYbGPpb-YUvF-2NZzODiQ1OvWQBRHSbPWYz_xbGkgD504LRtqRwCO7CC
_CyyURi1sEssPVsMJRX_U4LFEOc82TiDdqjKOjRUfKK5rqLi8nBE9soQ0DSaOoFQZi
GrBrqxDsNYiAYAmxxkos-i3nX4qtByVx85sCE5U_0MqG7COxZWMOPEFrDaepUV-cOy
rvoUIng8i8ljKBKxETY2BgPegKBYCxsAUcAkKamSCC9AiBxA0UOHyhTqtlvMksO7AE
hNC2-YzPyx1FkhMoS4LLe6E_pFsMlmjA6P1NSge9C5G5tETYXGAn6b1xZbHtmwrPSc
ro9LWhVmAaA7_bxYObnFUxgWtK4vzzQBjZJ36UTk4OTB-JvKWgfVWCFsaw5WCHj6Oo
4jpO7d2yN7WMfAj2hTEabz9wumQ0TMhBduZ-QON3pYObSy7TSC1vVme0NJrwF_cJRe
hKTFmdlXGVldPxZCplr7ZQqRQhF8JP-l4mEQVnCaWGn9ONHlemczGOS-A-wwtnmwjI
B1V_vgJRf4FdpV-4hUk4-QLpu3-1lWFxrtZKcggq3tWTduRo5_QebQbUUT_VSCgsFc
OmyWKoj56lbxthN19hq1XGWbLGfrrR6MWh23vk01zn8FVwi7uFwEnRYSafsnWLa1Z5
TpBj9GvAdl2H9NHwzpB5NqHpZNkQ3NMDj13Fn8fzO0JB83Etbm_tnFQfcb13X3bJ15
Cz-Ww1MGhvIpGGnMBT_ADp9xSIyAM9dQ1yeVXk-AIgWBUlN5uyWSGyCxp0cJwx7HxM
38z0UIeBu-MytL-eqndM7LxytsVzCbjOTSVRmhYEMIzUAnS1gs7uMQAGRdgRIElTJE
SGMjb_4bZq9s6Ve1LKkSi0_QDsrABaLe55UY0zF4ZSfOV5PMyPtocwV_dcNPlxLgNA
D1BFX_Z9kAdMZQW6fAmsfFle0zAoMe4l9pMESH0JB4sJGdCKtQXj1cXNydDYozF7l8
H00BV_Er7zd6VtIw0MxwkFCTatsv_R-GsBCH218RgVPsfYhwVuT8R4HarpzsDBufC4
r8_c8fc9Z278sQ081jFjOja6L2x0N_ImzFNXU6xwO-Ska-QeuvYZ3X_L31ZOX4Llp-
7QSfgDoHnOxFv1Xws-D5mDHD3zxOup2b2TppdKTZb9eW2vxUVviM8OI9atBfPKMGAO
v9omA-6vv5IxUH0-lWMiHLQ_g8vnswp-Jav0c4t6URVUzujNOoNd_CBGGVnHiJTCHl
88LQxsqLHHIu4Fz-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

- Added Security Considerations text about binding attributes to keys.
- Incorporated additional terms defined in the JWE spec by reference.

-40

- Clarified the definitions of UTF8(STRING) and ASCII(STRING).
- Stated that line breaks are for display purposes only in places where this disclaimer was needed and missing.
- 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

- Addressed Gen-ART review comments by Russ Housley.

- Addressed secdir review comments by Stephen Kent.

-31

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

-30

- Added references and cleaned up the reference syntax in a few places.

- Applied minor wording changes to the Security Considerations section.

-29

- 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

- Revised the introduction to the Security Considerations section.

- Refined the text about when applications using encrypted JWKs and JWK Sets would not need to use the "cty" header parameter.

-27

- Added an example JWK early in the draft.

- Described additional security considerations.

- Added the "x5t#S256" (X.509 Certificate SHA-256 Thumbprint) JWK member.

- Addressed a few editorial issues.

-26

- Referenced Section 6 of RFC 6125 for TLS server certificate identity validation.

- Deleted misleading non-normative phrase from the "use" description.

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

-25

- Updated WebCrypto reference to refer to W3C Last Call draft.

-24

- Corrected the authentication tag value in the encrypted key example.
- Updated the JSON reference to RFC 7159.

-23

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

-22

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

-21

- 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.
- 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.
- Changed some references from being normative to informative, addressing issue #90.

-20

- Renamed "use_details" to "key_ops" (key operations).
- 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.

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

-19

- Added optional "use_details" (key use details) JWK member.

- Reordered the key selection parameters.

-18

- Changes to address editorial and minor issues #68, #69, #73, #74, #76, #77, #78, #79, #82, #85, #89, and #135.

- Added and used Description registry fields.

-17

- Refined the "typ" and "cty" definitions to always be MIME Media Types, with the omission of "application/" prefixes recommended for brevity, addressing issue #50.

- Added an example encrypting an RSA private key with "PBES2-HS256+A128KW" and "A128CBC-HS256". Thanks to Matt Miller for producing this!

- Processing rules occurring in both JWS and JWK are now referenced in JWS by JWK, rather than duplicated, addressing issue #57.

- 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

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

-15

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

-14

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

-13

- Applied spelling and grammar corrections.

-12

- 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

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

-10

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

-09

- 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.
- Defined that members that are not understood must be ignored.

-08

- 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.
- Clarified statements of the form "This member is OPTIONAL" to "Use of this member is OPTIONAL".

- Referenced String Comparison Rules in JWS.

- Added seriesInfo information to Internet Draft references.

-07

- 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

- 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.)

- Clarify that the "alg" (algorithm family) member is REQUIRED.

- Correct an instance of "JWK" that should have been "JWK Set".

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

-05

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

-04

- Refer to the registries as the primary sources of defined values and then secondarily reference the sections defining the initial contents of the registries.

- Normatively reference XML DSIG 2.0 for its security considerations.

- Added this language to Registration Templates: "This name is case sensitive. Names that match other registered names in a case insensitive manner SHOULD NOT be accepted."

- Described additional open issues.

- Applied editorial suggestions.

-03

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

-02

- 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".
- Clarified that values with duplicate member names MUST be rejected.
- Established JSON Web Key Set Parameters registry.
- Explicitly listed non-goals in the introduction.
- Moved algorithm-specific definitions from JWK to JWA.
- Reformatted to give each member definition its own section heading.

-01

- Corrected the Magic Signatures reference.

-00

- 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

Table of Contents

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):

- JOSE Header
- JWS Payload
- 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):

- JWS Protected Header
- 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):

```
eyJpc3MiOiJqb2UiLA0KICJleHAiOjEzMDA4MTkzODAsDQogImh0dHA6Ly9leGFt
cGxlLmNvbS9pc19yb290Ijp0cnVlfQ
```

Computing the HMAC of the JWS Signing Input ASCII(BASE64URL(UTF8(JWS Protected Header)) || '.' || BASE64URL(JWS Payload)) with the HMAC

SHA-256 algorithm using the key specified in Appendix A.1 and base64url encoding the result yields this BASE64URL(JWS Signature) value:

```
dBjftJeZ4CVP-mB92K27uhbUJU1p1r_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
.
eyJpc3MiOiJqb2UiLA0KICJleHAiOjEzMDA4MTkzODAsDQogImh0dHA6Ly9leGFt
cGxlLmNvbS9pc19yb290Ijp0cnVlfQ
.
dBjftJeZ4CVP-mB92K27uhbUJU1p1r_wW1gFWFOEjXk
```

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 ECMAScript 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.

4.1.7. "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.

4.1.8. "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.

4.1.9. "typ" (Type) Header Parameter

The "typ" (type) Header Parameter is used by JWS applications to declare the MIME Media Type [IANA.MediaTypes] 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.MediaTypes] 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.

- Protocol Category: JSON Object Signing and Encryption (JOSE)
- Registry Location: http://www.iana.org/assignments/jose
- Webpage Title: (same as the protocol category)
- 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. 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.

9.1.2. Initial Registry Contents

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

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

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

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

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

- Header Parameter Name: "x5u"
- Header Parameter Description: X.509 URL
- Header Parameter Usage Location(s): JWS
- Change Controller: IESG
- Specification Document(s): Section 4.1.5 of [[this document]]

- Header Parameter Name: "x5c"
- Header Parameter Description: X.509 Certificate Chain
- Header Parameter Usage Location(s): JWS
- Change Controller: IESG
- Specification Document(s): Section 4.1.6 of [[this document]]

- Header Parameter Name: "x5t"
- Header Parameter Description: X.509 Certificate SHA-1 Thumbprint
- Header Parameter Usage Location(s): JWS
- Change Controller: IESG
- Specification Document(s): Section 4.1.7 of [[this document]]

- Header Parameter Name: "x5t#S256"
- Header Parameter Description: X.509 Certificate SHA-256 Thumbprint
- Header Parameter Usage Location(s): JWS
- Change Controller: IESG
- Specification Document(s): Section 4.1.8 of [[this document]]

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

- Header Parameter Name: "cty"
- Header Parameter Description: Content Type
- Header Parameter Usage Location(s): JWS
- Change Controller: IESG

- Specification Document(s): Section 4.1.10 of [[this document]]

- Header Parameter Name: "crit"
- Header Parameter Description: Critical
- Header Parameter Usage Location(s): JWS
- Change Controller: IESG
- 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.

- Type name: application
- Subtype name: jose
- Required parameters: n/a
- Optional parameters: n/a
- 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.
- Security considerations: See the Security Considerations section of [[this document]]
- Interoperability considerations: n/a
- Published specification: [[this document]]
- 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
- Fragment identifier considerations: n/a
- Additional information: Magic number(s): n/a, File extension(s): n/a, Macintosh file type code(s): n/a
- Person & email address to contact for further information: Michael B. Jones, mbj@microsoft.com
- Intended usage: COMMON
- Restrictions on usage: none
- Author: Michael B. Jones, mbj@microsoft.com
- Change Controller: IESG
- Provisional registration? No

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

- Verifiers of a signature support multiple algorithms.
- Given an existing signature, an attacker can find another payload that produces the same signature value with a different algorithm.
- The payload crafted by the attacker is valid in the application context.

There are several ways for an application to mitigate algorithm substitution attacks:

- 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.

- 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].)

- 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 ECMAScript 5.1 [ECMAScript]". 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 ECMAScript 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.MediaTypes] 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-xmldsig-bestpractices-20130411] Hirsch, F. and P. Datta, "XML Signature Best Practices", World Wide Web Consortium Note NOTE-xmldsig-bestpractices-20130411, April 2013, <http://www.w3.org/TR/2013/NOTE-xmldsig-bestpractices-20130411/>.

[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/>.

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":"AyM1SysPpbyDfgZld3umj1qzKObwVMkoqQ-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-mB92K27uhbUJU1p1r_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
.
eyJpc3MiOiJqb2UiLA0KICJleHAiOjEzMDA4MTkzODAsDQogImh0dHA6Ly9leGFt
cGxlLmNvbS9pc19yb290Ijp0cnVlfQ
.
dBjftJeZ4CVP-mB92K27uhbUJU1p1r_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
cGxlLmNvbS9pc19yb290Ijp0cnVlfQ
```

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_wWJcyQoTbji9k0l8W26mPddx
        HmfHQp-Vaw-4qPCJrcS2mJPMEzP1Pt0Bm4d4QlL-yRT-SFd2lZS-pCgNMs
        D1W_YpRPEwOWvG6b32690r2jZ47soMZo9wGzjb_7OMg0LOL-bSf63kpaSH
        SXndS5z5rexMdbBYUsLA9e-KXBdQOS-UTo7WTBEMa2R2CapHg665xsmtdV
        MTBQY4uDZlxvb3qCo5ZwKh9kG4LT6_I5IhlJH7aGhyxXFvUK-DWNmoudF8
        NAco9_h9iaGNj8q2ethFkMLs91kzk2PAcDTW9gb54h4FRWyuXpoQ",
   "e":"AQAB",
   "d":"Eq5xpGnNCivDflJsRQBXHx1hdR1k6Ulwe2JZD50LpXyWPEAeP88vLNO97I
        jlA7_GQ5sLKMgvfTeXZx9SE-7YwVol2NXOoAJe46sui395IW_GO-pWJ1O0
        BkTGoVEn2bKVRUCgu-GjBVaYLU6f3l9kJfFNS3E0QbVdxzubSu3Mkqzjkn
        439X0M_V51gfpRLI9JYanrC4D4qAdGcopV_0ZHHzQlBjudU2QvXt4ehNYT
        CBr6XCLQUShb1juUO1ZdiYoFaFQT5Tw8bGUl_x_jTj3ccPDVZFD9pIuhLh
        BOneufuBiB4cS98l2SR_RQyGWSeWjnczT0QU91p1DhOVRuOopznQ",
   "p":"4BzEEOtIpmVdVEZNCqS7baC4crd0pqnRH_5IB3jw3bcxGn6QLvnEtfdUdi
        YrqBdss1l58BQ3KhooKeQTa9AB0Hw_Py5PJdTJNPY8cQn7ouZ2KKDcmnPG
        BY5t7yLc1QlQ5xHdwW1VhvKn-nXqhJTBgIPgtldC-KDV5z-y2XDwGUc",
   "q":"uQPEfgmVtjL0Uyyx88GZFF1fOunH3-7cepKmtH4pxhtCoHqpWmT8YAmZxa
        ewHgHAjLYsp1ZSe7zFYHj7C6ul7TjeLQeZD_YwD66t62wDmpe_HlB-TnBA
        -njbglfIsRLtXlnDzQkv5dTltRJ11BKBBypeeF6689rjcJIDEz9RWdc",
   "dp":"BwKfV3Akq5_MFZDFZCnW-wzl-CCo83WoZvnLQwCTeDv8uzluRSnm71I3Q
        CLdhrqE2e9YkxvuxdBfpT_PI7Yz-FOKnu1R6HsJeDCjn12Sk3vmAktV2zb
        34MCdy7cpdTh_YVr7tss2u6vneTwrA86rZtu5Mbr1C1XsmvkxHQAdYo0",
   "dq":"h_96-mK1R_7glhsum81dZxjTnYynPbZpHziZjeeHcXYsXaaMwkOlODsWa
        7I9xXDoRwbKgB719rrmI2oKr6N3Do9U0ajaHF-NKJnwgjMd2w9cjz3_-ky
        NlxAr2v4IKhGNpmM5iIgOS1VZnOZ68m6_pbLBSp3nssTdlqvd0tIiTHU",
   "qi":"IYd7DHOhrWvxkwPQsRM2tOgrjbcrfvtQJipd-DlcxyVuuM9sQLdgjVk2o
        y26F0EmpScGLq2MowX7fhd_QJQ3ydy5cY7YIBi87w93IKLEdfnbJtoOPLU
        W0ITrJReOgo1cq9SbsxYawBgfp_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_O8W2Fp5jujGbds9uJdbF9CUAr7t1dnZcAcQjbKBYNX4
BAynRFdiuB--f_nZLgrnbyTyWzO75vRK5h6xBArLIARNPvkSjtQBMHlb1L07Qe7K
0GarZRmB_eSN9383LcOLn6_dO--xi12jzDwusC-eOkHWEsqtFZESc6BfI7noOPqv
hJ1phCnvWh6IeYI2w9QOYEUipUTI8np6LbgGY9Fs98rqVt5AXLIhWkWywlVmtVrB
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
cGxlLmNvbS9pc19yb290Ijp0cnVlfQ
.
cC4hiUPoj9Eetdgtv3hF80EGrhuB__dzERat0XF9g2VtQgr9PJbu3XOiZj5RZmh7
AAuHIm4Bh-0Qc_lF5YKt_O8W2Fp5jujGbds9uJdbF9CUAr7t1dnZcAcQjbKBYNX4
BAynRFdiuB--f_nZLgrnbyTyWzO75vRK5h6xBArLIARNPvkSjtQBMHlb1L07Qe7K
0GarZRmB_eSN9383LcOLn6_dO--xi12jzDwusC-eOkHWEsqtFZESc6BfI7noOPqv
hJ1phCnvWh6IeYI2w9QOYEUipUTI8np6LbgGY9Fs98rqVt5AXLIhWkWywlVmtVrB
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
cGxlLmNvbS9pc19yb290Ijp0cnVlfQ
```

The resulting JWS Signing Input value, which is the ASCII representation of above string, is the following octet sequence:

```
[101, 121, 74, 104, 98, 71, 99, 105, 79, 105, 74, 70, 85, 122, 73,
```

49, 78, 105, 74, 57, 46, 101, 121, 74, 112, 99, 51, 77, 105, 79, 105,
74, 113, 98, 50, 85, 105, 76, 65, 48, 75, 73, 67, 74, 108, 101, 72,
65, 105, 79, 106, 69, 122, 77, 68, 65, 52, 77, 84, 107, 122, 79, 68,
65, 115, 68, 81, 111, 103, 73, 109, 104, 48, 100, 72, 65, 54, 76,
121, 57, 108, 101, 71, 70, 116, 99, 71, 120, 108, 76, 109, 78, 118,
98, 83, 57, 112, 99, 49, 57, 121, 98, 50, 57, 48, 73, 106, 112, 48,
99, 110, 86, 108, 102, 81]

This example uses the elliptic curve key represented in JSON Web Key [JWK] format below:

```
{"kty":"EC",
 "crv":"P-256",
 "x":"f83OJ3D2xF1Bg8vub9tLe1gHMzV76e8Tus9uPHvRVEU",
 "y":"x_FEzRu9m36HLN_tue659LNpXW6pCyStikYjKIWI5a0",
 "d":"jpsQnnGQmL-YBIffH1136cspYG6-0iY7X1fCE9-E9LI"
}
```

The ECDSA private part d is then passed to an ECDSA signing function, which also takes the curve type, P-256, the hash type, SHA-256, and the JWS Signing Input as inputs. The result of the digital signature is the EC point (R, S), where R and S are unsigned integers. In this example, the R and S values, given as octet sequences representing big endian integers are:

Result Name	Value
R	[14, 209, 33, 83, 121, 99, 108, 72, 60, 47, 127, 21, 88, 7, 212, 2, 163, 178, 40, 3, 58, 249, 124, 126, 23, 129, 154, 195, 22, 158, 166, 101]
S	[197, 10, 7, 211, 140, 60, 112, 229, 216, 241, 45, 175, 8, 74, 84, 128, 166, 101, 144, 197, 242, 147, 80, 154, 143, 63, 127, 138, 131, 163, 84, 213]

The JWS Signature is the value R || S. Encoding the signature as BASE64URL(JWS Signature) produces this value (with line breaks for display purposes only):

```
DtEhU3ljbEg8L38VWAfUAqOyKAM6-Xx-F4GawxaepmXFCgfTjDxw5djxLa8ISlSA
pmWQxfKTUJqPP3-Kg6NU1Q
```

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
cGxlLmNvbS9pc19yb290Ijp0cnVlfQ
.
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:

```
eyJhbGciOiJFUzUxMiJ9.UGF5bG9hZA
```

The resulting JWS Signing Input value, which is the ASCII representation of above string, is the following octet sequence:

[101, 121, 74, 104, 98, 71, 99, 105, 79, 105, 74, 70, 85, 122, 85, 120, 77, 105, 74, 57, 46, 85, 71, 70, 53, 98, 71, 57, 104, 90, 65]

This example uses the elliptic curve key represented in JSON Web Key [JWK] format below (with line breaks within values for display purposes only):

```
{"kty":"EC",
 "crv":"P-521",
 "x":"AekpBQ8ST8a8VcfVOTNl353vSrDCLLJXmPk06wTjxrrjcBpXp5EOnYG_
      NjFZ6OvLFV1jSfS9tsz4qUxcWceqwQGk",
 "y":"ADSmRA43Z1DSNx_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-l192l60hp9wAHZFVJbLfD_UxMi70cwnZOYaRI1bKPWROc-mZZq
wqT2SI-KGDKB34XO0aw_7XdtAG8GaSwFKdCAPZgoXD2YBJZCPEX3xKpRwcdOO8Kp
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-l192l60hp9wAHZFVJbLfD_UxMi70cwnZOYaRI1bKPWROc-mZZq
wqT2SI-KGDKB34XO0aw_7XdtAG8GaSwFKdCAPZgoXD2YBJZCPEX3xKpRwcdOO8Kp
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:

```
eyJhbGciOiJub25lIn0
```

The JWS Payload used in this example, which follows, is the same as in the previous examples. Since the BASE64URL(JWS Payload) value will therefore be the same, its computation is not repeated here.

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

The JWS Signature is the empty octet string and BASE64URL(JWS Signature) is the empty string.

Concatenating these 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):

```
eyJhbGciOiJub25lIn0
.
eyJpc3MiOiJqb2UiLA0KICJleHAiOjEzMDA4MTkzODAsDQogImh0dHA6Ly9leGFt
cGxlLmNvbS9pc19yb290Ijp0cnVlfQ
.
```

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
cGxlLmNvbS9pc19yb290Ijp0cnVlfQ
```

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":
    "eyJpc3MiOiJqb2UiLA0KICJleHAiOjEzMDA4MTkzODAsDQogImh0dHA6Ly9leGF
     tcGxlLmNvbS9pc19yb290Ijp0cnVlfQ",
   "signatures":[
    {"protected":"eyJhbGciOiJSUzI1NiJ9",
     "header":
      {"kid":"2010-12-29"},
     "signature":
      "cC4hiUPoj9Eetdgtv3hF80EGrhuB__dzERat0XF9g2VtQgr9PJbu3XOiZj5RZ
       mh7AAuHIm4Bh-0Qc_lF5YKt_O8W2Fp5jujGbds9uJdbF9CUAr7t1dnZcAcQjb
       KBYNX4BAynRFdiuB--f_nZLgrnbyTyWzO75vRK5h6xBArLIARNPvkSjtQBMHl
       b1L07Qe7K0GarZRmB_eSN9383LcOLn6_dO--xi12jzDwusC-eOkHWEsqtFZES
       c6BfI7noOPqvhJ1phCnvWh6IeYI2w9QOYEUipUTI8np6LbgGY9Fs98rqVt5AX
       LIhWkWywlVmtVrBp0igcN_IoypGlUPQGe77Rw"},
    {"protected":"eyJhbGciOiJFUzI1NiJ9",
     "header":
      {"kid":"e9bc097a-ce51-4036-9562-d2ade882db0d"},
     "signature":
      "DtEhU3ljbEg8L38VWAfUAqOyKAM6-Xx-F4GawxaepmXFCgfTjDxw5djxLa8IS
       lSApmWQxfKTUJqPP3-Kg6NU1Q"}]
  }
```

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":
  "eyJpc3MiOiJqb2UiLA0KICJleHAiOjEzMDA4MTkzODAsDQogImh0dHA6Ly9leGF
   tcGxlLmNvbS9pc19yb290Ijp0cnVlfQ",
 "protected":"eyJhbGciOiJFUzI1NiJ9",
 "header":
  {"kid":"e9bc097a-ce51-4036-9562-d2ade882db0d"},
 "signature":
  "DtEhU3ljbEg8L38VWAfUAqOyKAM6-Xx-F4GawxaepmXFCgfTjDxw5djxLa8IS
   lSApmWQxfKTUJqPP3-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):

```
["MIIE3jCCA8agAwIBAgICAwEwDQYJKoZIhvcNAQEFBQAwYzELMAkGA1UEBhMCVVM
  xITAfBgNVBAoTGFRoZSBHbyBEYWRkeSBHcm91cCwgSW5jLjExMC8GA1UECxMoR2
  8gRGFkZHkgQ2xhc3MgMiBDZXJ0aWZpY2F0aW9uIEF1dGhvcml0eTAeFw0wNjExM
  TYwMTU0MzdaFw0yNjExMTYwMTU0MzdaMIHKMQswCQYDVQQGEwJVUzEQMA4GA1UE
  CBMHQXJpem9uYTETMBEGA1UEBxMKU2NvdHRzZGFsZTEaMBgGA1UEChMRR29EYWR
  keS5jb20sIEluYy4xMzAxBgNVBAsTKmh0dHA6Ly9jZXJ0aWZpY2F0ZXMuZ29kYW
  RkeS5jb20vcmVwb3NpdG9yeTEwMC4GA1UEAxMnR28gRGFkZHkgU2VjdXJlIENlc
  nRpZmljYXRpb24gQXV0aG yaXR5MREwDwYDVQQFEwgwNzk2OTI4NzCCASIwDQYJ
  KoZIhvcNAQEBBQADggEPADCCAQoCggEBAMQt1RWMnCZM7DI161+4WQFapmGBWTt
  wY6vj3D3HKrjJM9N55DrtPDAjhI6zMBS2sofDPZVUBJ7fmd0LJR4h3mUpfjWoqV
  Tr9vcyOdQmVZWt7/v+WIbXnvQAjYwqDL1CBM6nPwT27oDyqu9SoWlm2r4arV3aL
  GbqGmu75RpRSgAvSMeYddi5Kcju+GZtCpyz8/x4fKL4o/K1w/O5epHBp+YlLpyo
  7RJlbmr2EkRTcDCVw5wrWCs9CHRK8r5RsL+H0EwnWGu1NcWdrxcx+AuP7q2BNgW
  JCJjPOq8lh8BJ6qf9Z/dFjpfMFDniNoW1fho3/Rb2cRGadDAW/hOUoz+EDU8CAw
  EAAaOCATIwggEuMB0GA1UdDgQWBBT9rGEyk2xF1uLuhV+auud2mWjM5zAfBgNVH
  SMEGDAWgBTSxLDSkdRMEXGzYcs9of7dqGrU4zASBgNVHRMBAf8ECDAGAQH/AgEA
  MDMGCCsGAQUFBwEBBCcwJTAjBggrBgEFBQcwAYYXaHR0cDovL29jc3AuZ29kYWR
  keS5jb20wRgYDVR0fBD8wPTA7oDmgN4Y1aHR0cDovL2NlcnRpZmljYXRlcy5nb2
  RhZGR5LmNvbS9yZXBvc2l0b3J5L2dkcm9vdC5jcmwwSwYDVR0gBEQwQjBABgRVH
  SAAMDgwNgYIKwYBBQUHAgEWKmh0dHA6Ly9jZXJ0aWZpY2F0ZXMuZ29kYWRkeS5j
  b20vcmVwb3NpdG9yeTAOBgNVHQ8BAf8EBAMCAQYwDQYJKoZIhvcNAQEFBQADggE
  BANKGwOy9+aG2Z+5mC6IGOgRQjhVyrEp0lVPLN8tESe8HkGsz2ZbwlFalEzAFPI
  UyIXvJxwqoJKSQ3kbTJSMUA2fCENZvD117esyfxVgqwcSeIaha86ykRvOe5GPLL
  5CkKSkB2XIsKd83ASe8T+5o0yGPwLPk9Qnt0hCqU7S+8MxZC9Y7lhyVJEnfzuz9
  p0iRFEUOOjZv2kWzRaJBydTXRE4+uXR21aITVSzGh6O1mawGhId/dQb8vxRMDsx
  uxN89txJx9OjxUUAiKEngHUuHqDTMBqLdElrRhjZkAzVvb3du6/KFUJheqwNTrZ
  EjYx8WnM25sgVjOuH0aBsXBTWVU+4=",
 "MIIE+zCCBGSgAwIBAgICAQ0wDQYJKoZIhvcNAQEFBQAwgbsxJDAiBgNVBAcTG1Z
```

```
      hbGlDZXJ0IFZhbGlkYXRpb24gTmV0d29yazEXMBUGA1UEChMOVmFsaUNlcnQsIE
      luYy4xNTAzBgNVBAsTLFZhbGlDZXJ0IENsYXNzIDIgUG9saWN5IFZhbGlkYXRpb
      24gQXV0aG9yaXR5MSEwHwYDVQQDExhodHRwOi8vd3d3LnZhbGljZXJ0LmNvbS8x
      IDAeBgkqhkiG9w0BCQEWEWluZm9AdmFsaWNlcnQuY29tMB4XDTA0MDYyOTE3MDY
      yMFoXDTI0MDYyOTE3MDYyMFowYzELMAkGA1UEBhMCVVMxITAfBgNVBAoTGFRoZS
      BHbyBEYWRkeSBHcm91cCwgSW5jLjExMC8GA1UECxMoR28gRGFkZHkgQ2xhc3MgM
      iBDZXJ0aWZpY2F0aW9uIEF1dGhvcml0eTCCASAwDQYJKoZIhvcNAQEBBQADggEN
      ADCCAQgCggEBAN6d1+pXGEmhW+vXX0iG6r7d/+TvZxz0ZWizV3GgXne77ZtJ6XC
      APVYYYwhv2vLM0D9/AlQiVBDYsoHUwHU9S3/Hd8M+eKsaA7Ugay9qK7HFiH7Eux
      6wwdhFJ2+qN1j3hybX2C32qRe3H3I2TqYXP2WYktsqbl2i/ojgC95/5Y0V4evLO
      tXiEqITLdiOr18SPaAIBQi2XKVlOARFmR6jYGB0xUGlcmIbYsUfb18aQr4CUWWo
      riMYavx4A6lNf4DD+qta/KFApMoZFv6yyO9ecw3ud72a9nmYvLEHZ6IVDd2gWMZ
      Eewo+YihfukEHU1jPEX44dMX4/7VpkI+EdOqXG68CAQOjggHhMIIB3TAdBgNVHQ
      4EFgQU0sSw0pHUTBFxs2HLPaH+3ahq1OMwgdIGA1UdIwSByjCBx6GBwaSBvjCBu
      zEkMCIGA1UEBxMbVmFsaUNlcnQgVmFsaWRhdGlvbiBOZXR3b3JrMRcwFQYDVQQK
      Ew5WYWxpQ2VydCwgSW5jLjE1MDMGA1UECxMsVmFsaUNlcnQgQ2xhc3MgMiBQb2x
      pY3kgVmFsaWRhdGlvbiBBdXRob3JpdHkxITAfBgNVBAMTGGh0dHA6Ly93d3cudm
      FsaWNlcnQuY29tLzEgMB4GCSqGSIb3DQEJARYRaW5mb0B2YWxpY2VydC5jb22CA
      QEwDwYDVR0TAQH/BAUwAwEB/zAzBggrBgEFBQcBAQQnMCUwIwYIKwYBBQUHMAGG
      F2h0dHA6Ly9vY3NwLmdvZGFkZHkuY29tMEQGA1UdHwQ9MDswOaA3oDWGM2h0dHA
      6Ly9jZXJ0aWZpY2F0ZXMuZ29kYWRkeS5jb20vcmVwb3NpdG9yeS9yb290LmNybD
      BLBgNVHSAERDBCMEAGBFUdIAAwODA2BggrBgEFBQcCARYqaHR0cDovL2NlcnRpZ
      mljYXRlcy5nb2RhZGR5LmNvbS9yZXBvc2l0b3J5MA4GA1UdDwEB/wQEAwIBBjAN
      BgkqhkiG9w0BAQUFAAOBgQC1QPmnHfbq/qQaQlpE9xXUhUaJwL6e4+PrxeNYiY+
      Sn1eocSxI0YGyeR+sBjUZsE4OWBsUs5iB0QQeyAfJg594RAoYC5jcdnplDQ1tgM
      QLARzLrUc+cb53S8wGd9D0VmsfSxOaFIqII6hR8INMqzW/Rn453HWkrugp++85j
      09VZw==",
    "MIIC5zCCAlACAQEwDQYJKoZIhvcNAQEFBQAwgbsxJDAiBgNVBAcTG1ZhbGlDZXJ
      0IFZhbGlkYXRpb24gTmV0d29yazEXMBUGA1UEChMOVmFsaUNlcnQsIEluYy4xNT
      AzBgNVBAsTLFZhbGlDZXJ0IENsYXNzIDIgUG9saWN5IFZhbGlkYXRpb24gQXV0a
      G9yaXR5MSEwHwYDVQQDExhodHRwOi8vd3d3LnZhbGljZXJ0LmNvbS8xIDAeBgkq
      hkiG9w0BCQEWEWluZm9AdmFsaWNlcnQuY29tMB4XDTk5MDYyNjAwMTk1NFoXDTE
      5MDYyNjAwMTk1NFowgbsxJDAiBgNVBAcTG1ZhbGlDZXJ0IFZhbGlkYXRpb24gTm
      V0d29yazEXMBUGA1UEChMOVmFsaUNlcnQsIEluYy4xNTAzBgNVBAsTLFZhbGlDZ
      XJ0IENsYXNzIDIgUG9saWN5IFZhbGlkYXRpb24gQXV0aG9yaXR5MSEwHwYDVQQD
      ExhodHRwOi8vd3d3LnZhbGljZXJ0LmNvbS8xIDAeBgkqhkiG9w0BCQEWEWluZm9
      AdmFsaWNlcnQuY29tMIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQDOOnHK5a
      vIWZJV16vYdA757tn2VUdZZUcOBVXc65g2PFxTXdMwzzjsvUGJ7SVCCSRrCl6zf
      N1SLUzm1NZ9WlmpZdRJEy0kTRxQb7XBhVQ7/nHk01xC+YDgkRoKWzk2Z/M/VXwb
      P7RfZHM047QSv4dk+NoS/zcnwbNDu+97bi5p9wIDAQABMA0GCSqGSIb3DQEBBQU
      AA4GBADt/UG9vUJSZSWI4OB9L+KXIPqeCgfYrx+jFzug6EILLGACOTb2oWH+heQ
      C1u+mNr0HZDzTuIYEZoDJJKPTEjlbVUjP9UNV+mWwD5MlM/Mtsq2azSiGM5bUMM
      j4QssxsodyamEwCW/POuZ6lcg5Ktz885hZo+L7tdEy8W9ViH0Pd"]
```

Appendix C. Notes on implementing base64url encoding without padding

This appendix describes how to implement base64url encoding and decoding functions without padding based upon standard base64 encoding and decoding functions that do use padding.

To be concrete, example C# code implementing these functions is shown below. Similar code could be used in other languages.

```
static string base64urlencode(byte [] arg)
{
  string s = Convert.ToBase64String(arg); // Regular base64 encoder
  s = s.Split('=')[0]; // Remove any trailing '='s
  s = s.Replace('+', '-'); // 62nd char of encoding
  s = s.Replace('/', '_'); // 63rd char of encoding
  return s;
}

static byte [] base64urldecode(string arg)
{
  string s = arg;
  s = s.Replace('-', '+'); // 62nd char of encoding
  s = s.Replace('_', '/'); // 63rd char of encoding
  switch (s.Length % 4) // Pad with trailing '='s
  {
    case 0: break; // No pad chars in this case
    case 2: s += "=="; break; // Two pad chars
    case 3: s += "="; break; // One pad char
    default: throw new System.Exception(
      "Illegal base64url string!");
  }
  return Convert.FromBase64String(s); // Standard base64 decoder
}
```

As per the example code above, the number of '=' padding characters that needs to be added to the end of a base64url encoded string without padding to turn it into one with padding is a deterministic function of the length of the encoded string. Specifically, if the length mod 4 is 0, no padding is added; if the length mod 4 is 2, two '=' padding characters are added; if the length mod 4 is 3, one '=' padding character is added; if the length mod 4 is 1, the input is malformed.

An example correspondence between unencoded and encoded values follows. The octet sequence below encodes into the string below, which when decoded, reproduces the octet sequence.
3 236 255 224 193
A-z_4ME

Appendix D. 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

- Changed more instances of "reject" to "consider to be invalid".
- Simplified the wording of a Message Signature or MAC Computation step.

-40

- Clarified the definitions of UTF8(STRING) and ASCII(STRING).
- Stated that line breaks are for display purposes only in places where this disclaimer was needed and missing.

-39

- Updated the reference to draft-ietf-uta-tls-bcp.

-38

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

-37

- Updated the TLS requirements language to only require implementations to support TLS when they support features using

TLS.

- Updated the language about integrity protecting Header Parameters when used in a trust decision.
- Restricted algorithm names to using only ASCII characters.
- 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.
- Added the CRT parameter values to example RSA private key representations.
- Updated the example IANA registration request subject line.

-36

- Defined a flattened JWS JSON Serialization syntax, which is optimized for the single digital signature or MAC case.
- Clarified where white space and line breaks may occur in JSON objects by referencing Section 2 of RFC 7159.
- Specified that registration reviews occur on the jose-reg-review@ietf.org mailing list.

-35

- Addressed AppsDir reviews by Ray Polk.
- Used real values for examples in the IANA Registration Template.

-34

- Addressed IESG review comments by Alissa Cooper, Pete Resnick, Richard Barnes, Ted Lemon, and Stephen Farrell.
- Addressed Gen-ART review comments by Russ Housley.
- Referenced RFC 4945 for PEM certificate delimiter syntax.

-33

- Noted that certificate thumbprints are also sometimes known as certificate fingerprints.

- Added an informative reference to draft-ietf-uta-tls-bcp for recommendations on improving the security of software and services using TLS.
- Changed the registration review period to three weeks.
- Acknowledged additional contributors.

-32

- Addressed Gen-ART review comments by Russ Housley.
- Addressed secdir review comments by Tero Kivinen, Stephen Kent, and Scott Kelly.
- Replaced the term Plaintext JWS with Unsecured JWS.

-31

- Reworded the language about JWS implementations ignoring the "typ" and "cty" parameters, explicitly saying that their processing is performed by JWS applications.
- 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

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

-29

- 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

- 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.

- Added text about when applications typically would and would not use "typ" and "cty" header parameters.

-27

- Added the "x5t#S256" (X.509 Certificate SHA-256 Thumbprint) header parameter.

- Stated that any JSON inputs not conforming to the JSON-text syntax defined in RFC 7159 input MUST be rejected in their entirety.

- Simplified the TLS requirements.

-26

- Referenced Section 6 of RFC 6125 for TLS server certificate identity validation.

- 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.

- Added a small amount of additional explanatory text to the signature validation examples to aid implementers.

- Noted that octet sequences are depicted using JSON array notation.

- Updated references, including to W3C specifications.

-25

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

-24

- Updated the JSON reference to RFC 7159.

-23

- Clarified that the base64url encoding includes no line breaks, white space, or other additional characters.

-22

- Corrected RFC 2119 terminology usage.

- Replaced references to draft-ietf-json-rfc4627bis with RFC 7158.

-21

- Applied review comments to the appendix "Notes on Key Selection", addressing issue #93.

- Changed some references from being normative to informative, addressing issue #90.

- Applied review comments to the JSON Serialization section, addressing issue #121.

-20

- Made terminology definitions more consistent, addressing issue #165.

- Restructured the JSON Serialization section to call out the parameters used in hanging lists, addressing issue #121.

- Described key filtering and refined other aspects of the text in the appendix "Notes on Key Selection", addressing issue #93.

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

-19

- Added the appendix "Notes on Validation Key Selection", addressing issue #93.

- Reordered the key selection parameters.

-18

- 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.

- Changes to address editorial and minor issues #25, #89, #97, #110, #114, #115, #116, #117, #120, and #184.

- Added and used Header Parameter Description registry field.

-17

- Refined the "typ" and "cty" definitions to always be MIME Media Types, with the omission of "application/" prefixes recommended for brevity, addressing issue #50.

- 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.

- 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.

- Replaced verbose repetitive phases such as "base64url encode the octets of the UTF-8 representation of X" with mathematical notation such as "BASE64URL(UTF8(X))".

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

-16

- Changes to address editorial and minor issues #50, #98, #99, #102, #104, #106, #107, #111, and #112.

-15

- 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.

- Corrected editorial error in "ES512" example.

- Changes to address editorial and minor issues #34, #96, #100, #101, #104, #105, and #106.

-14

- 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

- Made all header parameter values be per-signature/MAC, addressing issue #24.

-12

- Clarified that the "typ" and "cty" header parameters are used in an application-specific manner and have no effect upon the JWS processing.

- Replaced the MIME types "application/jws+json" and "application/jws" with "application/jose+json" and "application/jose".

- 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.

- 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

- Added Key Identification section.

- 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.)

- Removed suggested compact serialization for multiple digital signatures and/or MACs.

- Changed the MIME type name "application/jws-js" to "application/jws+json", addressing issue #22.

- Tightened the description of the "crit" (critical) header parameter.

- Added a negative test case for the "crit" header parameter

-10

- Added an appendix suggesting a possible compact serialization for JWSs with multiple digital signatures and/or MACs.

-09

- Added JWS JSON Serialization, as specified by draft-jones-jose-jws-json-serialization-04.
- Registered "application/jws-js" MIME type and "JWS-JS" typ header parameter value.
- 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.
- Changed term "JWS Secured Input" to "JWS Signing Input".
- Changed from using the term "byte" to "octet" when referring to 8 bit values.
- Changed member name from "recipients" to "signatures" in the JWS JSON Serialization.
- Added complete values using the JWS Compact Serialization for all examples.

-08

- Applied editorial improvements suggested by Jeff Hodges and Hannes Tschofenig. Many of these simplified the terminology used.
- Clarified statements of the form "This header parameter is OPTIONAL" to "Use of this header parameter is OPTIONAL".
- Added a Header Parameter Usage Location(s) field to the IANA JSON Web Signature and Encryption Header Parameters registry.
- Added seriesInfo information to Internet Draft references.

-07

- Updated references.

-06

- 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.

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

-05

- Added statement that "StringOrURI values are compared as case-sensitive strings with no transformations or canonicalizations applied".

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

-04

- Completed JSON Security Considerations section, including considerations about rejecting input with duplicate member names.

- Completed security considerations on the use of a SHA-1 hash when computing "x5t" (x.509 certificate thumbprint) values.

- Refer to the registries as the primary sources of defined values and then secondarily reference the sections defining the initial contents of the registries.

- Normatively reference XML DSIG 2.0 for its security considerations.

- 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."

- Reference draft-jones-jose-jws-json-serialization instead of draft-jones-json-web-signature-json-serialization.

- Described additional open issues.

- Applied editorial suggestions.

-03

- 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.

- Added "Collision Resistant Namespace" to the terminology section.
- Reference ITU.X690.1994 for DER encoding.
- 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.
- Added an example "x5c" value.
- No longer say "the UTF-8 representation of the JWS Secured Input (which is the same as the ASCII representation)". Just call it "the ASCII representation of the JWS Secured Input".
- Added Registration Template sections for defined registries.
- Added Registry Contents sections to populate registry values.
- 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.
- Moved registries JSON Web Signature and Encryption Header Parameters and JSON Web Signature and Encryption Type Values to the JWS specification.
- Numerous editorial improvements.

-02

- Clarified that it is an error when a "kid" value is included and no matching key is found.
- Removed assumption that "kid" (key ID) can only refer to an asymmetric key.
- Clarified that JWSs with duplicate Header Parameter Names MUST be rejected.
- Clarified the relationship between "typ" header parameter values and MIME types.
- Registered application/jws MIME type and "JWS" typ header parameter value.

- 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).

- Added suggestion on defining additional header parameters such as "x5t#S256" in the future for certificate thumbprints using hash algorithms other than SHA-1.

- Specify RFC 2818 server identity validation, rather than RFC 6125 (paralleling the same decision in the OAuth specs).

- 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.

- Reformatted to give each header parameter its own section heading.

-01

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

- Added "jpk" and "x5c" header parameters for including JWK public keys and X.509 certificate chains directly in the header.

- Clarified that this specification is defining the JWS Compact Serialization. Referenced the new JWS-JS spec, which defines the JWS JSON Serialization.

- Added text "New header parameters should be introduced sparingly since an implementation that does not understand a parameter MUST reject the JWS".

- 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.

- Changed "no canonicalization is performed" to "no canonicalization need be performed".

- Corrected the Magic Signatures reference.

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

-00

- Created the initial IETF draft based upon draft-jones-json-web-signature-04 with no normative changes.

- 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/