

JOSE
Internet-Draft
Intended status: Informational
Expires: April 25, 2013

R. Barnes
BBN Technologies
October 22, 2012

Use Cases and Requirements for JSON Object Signing and Encryption (JOSE)
draft-barnes-jose-use-cases-01.txt

Abstract

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

Status of this Memo

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

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

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

This Internet-Draft will expire on April 25, 2013.

Copyright Notice

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

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

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

Table of Contents

1. Introduction	3
2. Definitions	4
3. Basic Requirements	5
4. Use Cases	5
4.1. Security Tokens and Authorization	6
4.2. XMPP	8
4.3. ALTO	11
4.4. Emergency Alerting	11
5. Other Requirements	13
6. Acknowledgements	13
7. IANA Considerations	13
8. Security Considerations	13
9. References	14
9.1. Normative References	14
9.2. Informative References	15
Author's Address	16

1. Introduction

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

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

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

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

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

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

2. Definitions

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

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

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

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

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

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

3. Basic Requirements

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

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

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

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

4. Use Cases

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

requirements that they imply for the JOSE object formats.

4.1. Security Tokens and Authorization

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

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

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

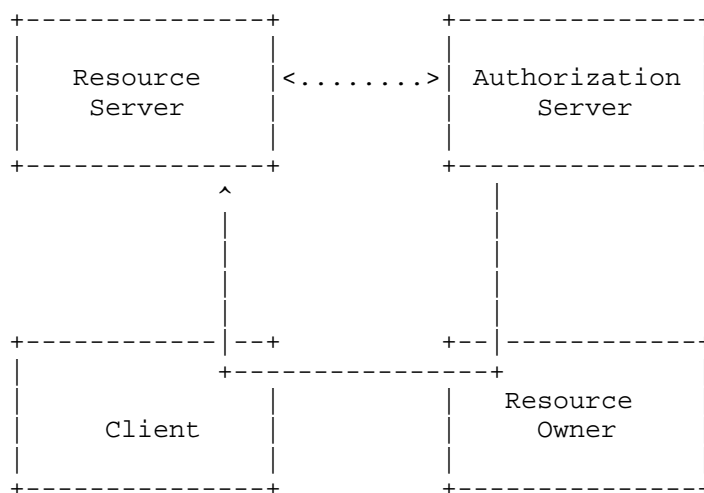


Figure 1: The OAuth process

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

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

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

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

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

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

Two related security token systems have similar requirements:

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

4.2. XMPP

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

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

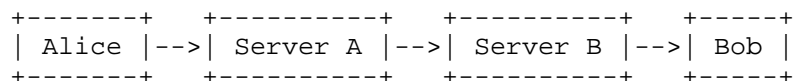


Figure 2: Delivering an XMPP message

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

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

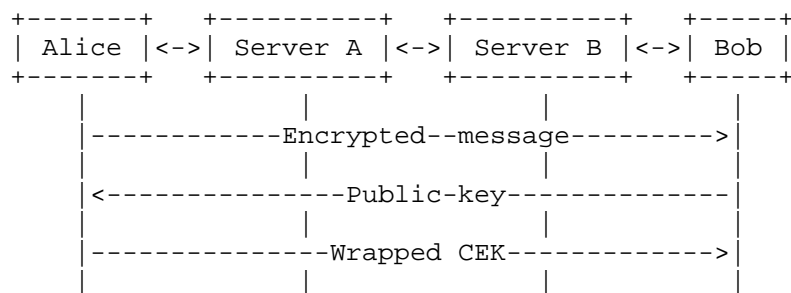


Figure 3: Delivering a secure XMPP message

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

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

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

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

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

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

4.3. ALTO

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

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

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

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

4.4. Emergency Alerting

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

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

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

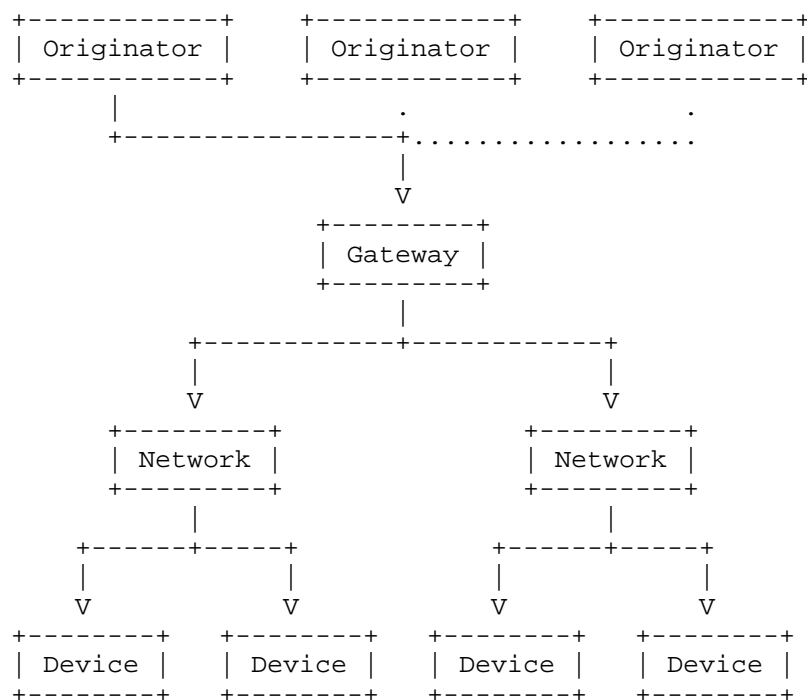


Figure 4: Delivering an emergency alert

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

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

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

5. Other Requirements

[[For the initial version of this document, this section is a placeholder, to incorporate any further requirements not directly derived from the above use cases.]]

6. Acknowledgements

Thanks to Matt Miller for discussions related to XMPP end-to-end security model, and to Mike Jones for considerations related to security tokens and XML security.

7. IANA Considerations

This document makes no request of IANA.

8. Security Considerations

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

9. References

9.1. Normative References

- [I-D.ietf-alto-protocol]
Alimi, R., Penno, R., and Y. Yang, "ALTO Protocol", draft-ietf-alto-protocol-13 (work in progress), September 2012.
- [I-D.ietf-alto-reqs]
Kiesel, S., Previdi, S., Stiernerling, M., Woundy, R., and Y. Yang, "Application-Layer Traffic Optimization (ALTO) Requirements", draft-ietf-alto-reqs-16 (work in progress), June 2012.
- [I-D.ietf-atoca-requirements]
Schulzrinne, H., Norreys, S., Rosen, B., and H. Tschofenig, "Requirements, Terminology and Framework for Exigent Communications", draft-ietf-atoca-requirements-03 (work in progress), March 2012.
- [I-D.ietf-oauth-json-web-token]
Jones, M., Bradley, J., and N. Sakimura, "JSON Web Token (JWT)", draft-ietf-oauth-json-web-token-04 (work in progress), October 2012.
- [I-D.miller-xmpp-e2e]
Miller, M., "End-to-End Object Encryption for the Extensible Messaging and Presence Protocol (XMPP)", draft-miller-xmpp-e2e-02 (work in progress), July 2012.
- [RFC4627] Crockford, D., "The application/json Media Type for JavaScript Object Notation (JSON)", RFC 4627, July 2006.
- [RFC4648] Josefsson, S., "The Base16, Base32, and Base64 Data Encodings", RFC 4648, October 2006.
- [RFC4949] Shirey, R., "Internet Security Glossary, Version 2", RFC 4949, August 2007.
- [RFC6120] Saint-Andre, P., "Extensible Messaging and Presence Protocol (XMPP): Core", RFC 6120, March 2011.
- [RFC6749] Hardt, D., "The OAuth 2.0 Authorization Framework", RFC 6749, October 2012.
- [W3C.CR-xmlsig-core2-20120124]
Solo, D., Eastlake, D., Hirsch, F., Reagle, J., Roessler, T., Datta, P., Cantor, S., and K. Yiu, "XML Signature Syntax and Processing Version 2.0", World Wide Web

Consortium CR CR-xmlsig-core2-20120124, January 2012,
<<http://www.w3.org/TR/2012/CR-xmlsig-core2-20120124>>.

[W3C.CR-xmlenc-core1-20120313]
Eastlake, D., Reagle, J., Roessler, T., and F. Hirsch,
"XML Encryption Syntax and Processing Version 1.1", World
Wide Web Consortium CR CR-xmlenc-core1-20120313,
March 2012,
<<http://www.w3.org/TR/2012/CR-xmlenc-core1-20120313>>.

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

9.2. Informative References

- [CAP] Botterell, A. and E. Jones, "Common Alerting Protocol
v1.1", October 2005.
- [I-D.ietf-oauth-jwt-bearer]
Jones, M., Campbell, B., and C. Mortimore, "JSON Web Token
(JWT) Bearer Token Profiles for OAuth 2.0",
draft-ietf-oauth-jwt-bearer-02 (work in progress),
September 2012.
- [I-D.ietf-oauth-saml2-bearer]
Campbell, B. and C. Mortimore, "SAML 2.0 Bearer Assertion
Profiles for OAuth 2.0", draft-ietf-oauth-saml2-bearer-14
(work in progress), September 2012.
- [ITU.X690.1994]
International Telecommunications Union, "Information
Technology - ASN.1 encoding rules: Specification of Basic
Encoding Rules (BER), Canonical Encoding Rules (CER) and
Distinguished Encoding Rules (DER)", ITU-T Recommendation
X.690, 1994.
- [OASIS.saml-core-2.0-os]
Cantor, S., Kemp, J., Philpott, R., and E. Maler,
"Assertions and Protocol for the OASIS Security Assertion
Markup Language (SAML) V2.0", OASIS Standard saml-core-
2.0-os, March 2005.
- [OpenID.Messages]
Sakimura, N., Bradley, J., Jones, M., de Medeiros, B.,
Mortimore, C., and E. Jay, "OpenID Connect Messages 1.0",

June 2012,
<[http://openid.net/specs/
openid-connect-messages-1_0.html](http://openid.net/specs/openid-connect-messages-1_0.html)>.

- [RFC2616] Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., and T. Berners-Lee, "Hypertext Transfer Protocol -- HTTP/1.1", RFC 2616, June 1999.
- [RFC2818] Rescorla, E., "HTTP Over TLS", RFC 2818, May 2000.
- [RFC3207] Hoffman, P., "SMTP Service Extension for Secure SMTP over Transport Layer Security", RFC 3207, February 2002.
- [RFC3923] Saint-Andre, P., "End-to-End Signing and Object Encryption for the Extensible Messaging and Presence Protocol (XMPP)", RFC 3923, October 2004.
- [RFC4301] Kent, S. and K. Seo, "Security Architecture for the Internet Protocol", RFC 4301, December 2005.
- [RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, August 2008.
- [RFC5322] Resnick, P., Ed., "Internet Message Format", RFC 5322, October 2008.
- [RFC5652] Housley, R., "Cryptographic Message Syntax (CMS)", STD 70, RFC 5652, September 2009.
- [RFC5751] Ramsdell, B. and S. Turner, "Secure/Multipurpose Internet Mail Extensions (S/MIME) Version 3.2 Message Specification", RFC 5751, January 2010.
- [WS-Federation] Kaler, C., McIntosh, M., Goodner, M., and A. Nadalin, "OpenID Connect Messages 1.0", May 2009, <<http://docs.oasis-open.org/wsfed/federation/v1.2/os/ws-federation-1.2-spec-os.html>>.

Author's Address

Richard Barnes
BBN Technologies
9861 Broken Land Parkway
Columbia, MD 21046
US

Phone: +1 410 290 6169
Email: rbarnes@bbn.com

JOSE Working Group
Internet-Draft
Intended status: Standards Track
Expires: April 18, 2013

M. Jones
Microsoft
October 15, 2012

JSON Web Algorithms (JWA)
draft-ietf-jose-json-web-algorithms-06

Abstract

The JSON Web Algorithms (JWA) specification enumerates cryptographic algorithms and identifiers to be used with the JSON Web Signature (JWS), JSON Web Encryption (JWE), and JSON Web Key (JWK) specifications.

Status of this Memo

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

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

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

This Internet-Draft will expire on April 18, 2013.

Copyright Notice

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

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

Table of Contents

1. Introduction	4
1.1. Notational Conventions	4
2. Terminology	4
2.1. Terms Incorporated from the JWS Specification	4
2.2. Terms Incorporated from the JWE Specification	5
2.3. Terms Incorporated from the JWK Specification	6
2.4. Defined Terms	7
3. Cryptographic Algorithms for JWS	7
3.1. "alg" (Algorithm) Header Parameter Values for JWS	7
3.2. MAC with HMAC SHA-256, HMAC SHA-384, or HMAC SHA-512	8
3.3. Digital Signature with RSA SHA-256, RSA SHA-384, or RSA SHA-512	9
3.4. Digital Signature with ECDSA P-256 SHA-256, ECDSA P-384 SHA-384, or ECDSA P-521 SHA-512	10
3.5. Using the Algorithm "none"	12
3.6. Additional Digital Signature/MAC Algorithms and Parameters	12
4. Cryptographic Algorithms for JWE	12
4.1. "alg" (Algorithm) Header Parameter Values for JWE	12
4.2. "enc" (Encryption Method) Header Parameter Values for JWE	14
4.3. Key Encryption with RSAES-PKCS1-V1_5	15
4.4. Key Encryption with RSAES OAEP	15
4.5. Key Encryption with AES Key Wrap	15
4.6. Direct Encryption with a Shared Symmetric Key	15
4.7. Key Agreement with Elliptic Curve Diffie-Hellman Ephemeral Static (ECDH-ES)	15
4.7.1. Key Derivation for "ECDH-ES"	16
4.8. Composite Plaintext Encryption Algorithms "A128CBC+HS256" and "A256CBC+HS512"	17
4.8.1. Key Derivation for "A128CBC+HS256" and "A256CBC+HS512"	17
4.8.2. Encryption Calculation for "A128CBC+HS256" and "A256CBC+HS512"	18
4.8.3. Integrity Calculation for "A128CBC+HS256" and "A256CBC+HS512"	18
4.9. Plaintext Encryption with AES GCM	19
4.10. Additional Encryption Algorithms and Parameters	19
5. Cryptographic Algorithms for JWK	20
5.1. "alg" (Algorithm Family) Parameter Values for JWK	20
5.2. JWK Parameters for Elliptic Curve Keys	21
5.2.1. "crv" (Curve) Parameter	21
5.2.2. "x" (X Coordinate) Parameter	21
5.2.3. "y" (Y Coordinate) Parameter	21
5.3. JWK Parameters for RSA Keys	22
5.3.1. "mod" (Modulus) Parameter	22

5.3.2. "xpo" (Exponent) Parameter	22
5.4. Additional Key Algorithm Families and Parameters	22
6. IANA Considerations	22
6.1. JSON Web Signature and Encryption Algorithms Registry . .	23
6.1.1. Registration Template	23
6.1.2. Initial Registry Contents	24
6.2. JSON Web Key Algorithm Families Registry	27
6.2.1. Registration Template	27
6.2.2. Initial Registry Contents	28
6.3. JSON Web Key Parameters Registration	28
6.3.1. Registry Contents	28
7. Security Considerations	29
8. References	30
8.1. Normative References	30
8.2. Informative References	31
Appendix A. Digital Signature/MAC Algorithm Identifier Cross-Reference	32
Appendix B. Encryption Algorithm Identifier Cross-Reference . . .	34
Appendix C. Acknowledgements	36
Appendix D. Open Issues	37
Appendix E. Document History	37
Author's Address	40

1. Introduction

The JSON Web Algorithms (JWA) specification enumerates 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. All these specifications utilize JavaScript Object Notation (JSON) [RFC4627] based data structures. This specification also describes the semantics and operations that are specific to these algorithms and algorithm families.

Enumerating the algorithms and identifiers for them in this specification, 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.

1.1. Notational Conventions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in Key words for use in RFCs to Indicate Requirement Levels [RFC2119].

2. Terminology

2.1. Terms Incorporated from the JWS Specification

These terms defined by the JSON Web Signature (JWS) [JWS] specification are incorporated into this specification:

JSON Web Signature (JWS) A data structure cryptographically securing a JWS Header and a JWS Payload with a JWS Signature value.

JWS Header A string representing a JavaScript Object Notation (JSON) [RFC4627] object that describes the digital signature or MAC operation applied to create the JWS Signature value.

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

JWS Signature A byte array containing the cryptographic material that secures the contents of the JWS Header and the JWS Payload.

Base64url Encoding The URL- and filename-safe Base64 encoding described in RFC 4648 [RFC4648], Section 5, with the (non URL-safe) '=' padding characters omitted, as permitted by Section 3.2. (See Appendix C of [JWS] for notes on implementing base64url

encoding without padding.)

Encoded JWS Header Base64url encoding of the bytes of the UTF-8 [RFC3629] representation of the JWS Header.

Encoded JWS Payload Base64url encoding of the JWS Payload.

Encoded JWS Signature Base64url encoding of the JWS Signature.

JWS Secured Input The concatenation of the Encoded JWS Header, a period ('.') character, and the Encoded JWS Payload.

Collision Resistant Namespace A namespace that allows names to be allocated in a manner such that they are highly unlikely to collide with other names. For instance, collision resistance can be achieved through administrative delegation of portions of the namespace or through use of collision-resistant name allocation functions. 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.

2.2. Terms Incorporated from the JWE Specification

These terms defined by the JSON Web Encryption (JWE) [JWE] specification are incorporated into this specification:

JSON Web Encryption (JWE) A data structure representing an encrypted version of a Plaintext. The structure consists of four parts: the JWE Header, the JWE Encrypted Key, the JWE Ciphertext, and the JWE Integrity Value.

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

Ciphertext The encrypted version of the Plaintext.

Content Encryption Key (CEK) A symmetric key used to encrypt the Plaintext for the recipient to produce the Ciphertext.

Content Integrity Key (CIK) A key used with a MAC function to ensure the integrity of the Ciphertext and the parameters used to create it.

Content Master Key (CMK) A key from which the CEK and CIK are derived. When key wrapping or key encryption are employed, the CMK is randomly generated and encrypted to the recipient as the JWE Encrypted Key. When key agreement is employed, the CMK is the result of the key agreement algorithm.

JWE Header A string representing a JSON object that describes the encryption operations applied to create the JWE Encrypted Key, the JWE Ciphertext, and the JWE Integrity Value.

JWE Encrypted Key When key wrapping or key encryption are employed, the Content Master Key (CMK) is encrypted with the intended recipient's key and the resulting encrypted content is recorded as a byte array, which is referred to as the JWE Encrypted Key. Otherwise, when key agreement is employed, the JWE Encrypted Key is the empty byte array.

JWE Ciphertext A byte array containing the Ciphertext.

JWE Integrity Value A byte array containing a MAC value that ensures the integrity of the Ciphertext and the parameters used to create it.

Encoded JWE Header Base64url encoding of the bytes of the UTF-8 [RFC3629] representation of the JWE Header.

Encoded JWE Encrypted Key Base64url encoding of the JWE Encrypted Key.

Encoded JWE Ciphertext Base64url encoding of the JWE Ciphertext.

Encoded JWE Integrity Value Base64url encoding of the JWE Integrity Value.

AEAD Algorithm An Authenticated Encryption with Associated Data (AEAD) [RFC5116] encryption algorithm is one that provides an integrated content integrity check. AES Galois/Counter Mode (GCM) is one such algorithm.

2.3. Terms Incorporated from the JWK Specification

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

JSON Web Key (JWK) A JSON data structure that represents a public key.

JSON Web Key Set (JWK Set) A JSON object that contains an array of JWKs as a member.

2.4. Defined Terms

These terms are defined for use by this specification:

Header Parameter Name The name of a member of the JSON object representing a JWS Header or JWE Header.

Header Parameter Value The value of a member of the JSON object representing a JWS Header or JWE Header.

3. Cryptographic Algorithms for JWS

JWS uses cryptographic algorithms to digitally sign or create a Message Authentication Codes (MAC) of the contents of the JWS Header and the JWS Payload. The use of the following algorithms for producing JWSs is defined in this section.

3.1. "alg" (Algorithm) Header Parameter Values for JWS

The table below is the set of "alg" (algorithm) header parameter values defined by this specification for use with JWS, each of which is explained in more detail in the following sections:

alg Parameter Value	Digital Signature or MAC Algorithm	Implementation Requirements
HS256	HMAC using SHA-256 hash algorithm	REQUIRED
HS384	HMAC using SHA-384 hash algorithm	OPTIONAL
HS512	HMAC using SHA-512 hash algorithm	OPTIONAL
RS256	RSASSA using SHA-256 hash algorithm	RECOMMENDED
RS384	RSASSA using SHA-384 hash algorithm	OPTIONAL
RS512	RSASSA using SHA-512 hash algorithm	OPTIONAL

ES256	ECDSA using P-256 curve and SHA-256 hash algorithm	RECOMMENDED+
ES384	ECDSA using P-384 curve and SHA-384 hash algorithm	OPTIONAL
ES512	ECDSA using P-521 curve and SHA-512 hash algorithm	OPTIONAL
none	No digital signature or MAC value included	REQUIRED

All the names are short because a core goal of JWS is for the representations to be compact. However, there is no a priori length restriction on "alg" values.

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

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

3.2. MAC with HMAC SHA-256, HMAC SHA-384, or HMAC SHA-512

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 the MAC matches the hashed content, in this case the JWS Secured Input, which therefore demonstrates that whoever generated the MAC was in possession of the secret. The means of exchanging the shared key is outside the scope of this specification.

The algorithm for implementing and validating HMACs is provided in RFC 2104 [RFC2104]. This section defines the use of the HMAC SHA-256, HMAC SHA-384, and HMAC SHA-512 functions [SHS]. The "alg" (algorithm) header parameter values "HS256", "HS384", and "HS512" are used in the JWS Header to indicate that the Encoded JWS Signature contains a base64url encoded HMAC value using the respective hash function.

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

The HMAC SHA-256 MAC is generated per RFC 2104, using SHA-256 as the hash algorithm "H", using the bytes of the ASCII [USASCII] representation of the JWS Secured Input as the "text" value, and using the shared key. The HMAC output value is the JWS Signature. The JWS signature is base64url encoded to produce the Encoded JWS

Signature.

The HMAC SHA-256 MAC for a JWS is validated by computing an HMAC value per RFC 2104, using SHA-256 as the hash algorithm "H", using the bytes of the ASCII representation of the received JWS Secured 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. Alternatively, the computed HMAC value can be base64url encoded and compared to the received Encoded JWS Signature, as this comparison produces the same result as comparing the unencoded values. In either case, if the values match, the HMAC has been validated. If the validation fails, the JWS MUST be rejected.

Securing content with the HMAC SHA-384 and HMAC SHA-512 algorithms is performed identically to the procedure for HMAC SHA-256 - just using the corresponding hash algorithm 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 RSA SHA-256, RSA SHA-384, or RSA SHA-512

This section defines the use of the RSASSA-PKCS1-V1_5 digital signature algorithm as defined in Section 8.2 of RFC 3447 [RFC3447], (commonly known as PKCS #1), using SHA-256, SHA-384, or SHA-512 [SHS] as the hash functions. The "alg" (algorithm) header parameter values "RS256", "RS384", and "RS512" are used in the JWS Header to indicate that the Encoded JWS Signature contains a base64url encoded RSA digital signature using the respective hash function.

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

The RSA SHA-256 digital signature is generated as follows:

1. Generate a digital signature of the bytes of the ASCII representation of the JWS Secured Input using RSASSA-PKCS1-V1_5-SIGN and the SHA-256 hash function with the desired private key. The output will be a byte array.
2. Base64url encode the resulting byte array.

The output is the Encoded JWS Signature for that JWS.

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

1. Take the Encoded JWS Signature and base64url decode it into a byte array. If decoding fails, the JWS MUST be rejected.
2. Submit the bytes of the ASCII representation of the JWS Secured Input 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.
3. If the validation fails, the JWS MUST be rejected.

Signing with the RSA SHA-384 and RSA SHA-512 algorithms is performed identically to the procedure for RSA SHA-256 - just using the corresponding hash algorithm with correspondingly larger result values: 384 bits for RSA SHA-384 and 512 bits for RSA SHA-512.

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

- 3.4. Digital Signature with ECDSA P-256 SHA-256, ECDSA P-384 SHA-384, or ECDSA P-521 SHA-512

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

This specification defines the use of ECDSA with the P-256 curve and the SHA-256 cryptographic hash function, ECDSA with the P-384 curve and the SHA-384 hash function, and ECDSA with the P-521 curve and the SHA-512 hash function. The P-256, P-384, and P-521 curves are defined in [DSS]. The "alg" (algorithm) header parameter values "ES256", "ES384", and "ES512" are used in the JWS Header to indicate that the Encoded JWS Signature contains a base64url encoded ECDSA P-256 SHA-256, ECDSA P-384 SHA-384, or ECDSA P-521 SHA-512 digital signature, respectively.

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

1. Generate a digital signature of the bytes of the ASCII representation of the JWS Secured 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 byte arrays in big endian order, with each array being 32 bytes long. The array representations MUST not be shortened to omit any leading zero bytes contained in the values.

3. Concatenate the two byte arrays in the order R and then S. (Note that many ECDSA implementations will directly produce this concatenation as their output.)
4. Base64url encode the resulting 64 byte array.

The output is the Encoded JWS Signature for the JWS.

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

1. Take the Encoded JWS Signature and base64url decode it into a byte array. If decoding fails, the JWS MUST be rejected.
2. The output of the base64url decoding MUST be a 64 byte array. If decoding does not result in a 64 byte array, the JWS MUST be rejected.
3. Split the 64 byte array into two 32 byte arrays. The first array will be R and the second S (with both being in big endian byte order).
4. Submit the bytes of the ASCII representation of the JWS Secured Input R, S and the public key (x, y) to the ECDSA P-256 SHA-256 validator.
5. If the validation fails, the JWS MUST be rejected.

Note that ECDSA digital signature contains a value referred to as K, which is a random number generated for each digital signature instance. This means that two ECDSA digital signatures using exactly the same input parameters will output different signature values because their K values will be different. A consequence of this is that one cannot validate an ECDSA signature by recomputing the signature and comparing the results.

Signing with the ECDSA P-384 SHA-384 and ECDSA P-521 SHA-512 algorithms is performed identically to the procedure for ECDSA P-256 SHA-256 - just using the corresponding hash algorithm with correspondingly larger result values. For ECDSA P-384 SHA-384, R and S will be 384 bits each, resulting in a 96 byte array. For ECDSA P-521 SHA-512, R and S will be 521 bits each, resulting in a 132 byte array.

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

3.5. Using the Algorithm "none"

JWSs MAY also be created that do not provide integrity protection. Such a JWS is called a "Plaintext JWS". Plaintext JWSs MUST use the "alg" value "none", and are formatted identically to other JWSs, but with an empty JWS Signature value.

3.6. Additional Digital Signature/MAC Algorithms and Parameters

Additional algorithms MAY be used to protect JWSs with corresponding "alg" (algorithm) header parameter values being defined to refer to them. New "alg" header parameter values SHOULD either be registered in the IANA JSON Web Signature and Encryption Algorithms registry Section 6.1 or be a URI that contains a Collision Resistant Namespace. In particular, it is permissible to use the algorithm identifiers defined in XML DSIG [RFC3275], XML DSIG 2.0 [W3C.CR-xmlsig-core2-20120124], and related specifications as "alg" values.

As indicated by the common registry, JWSs and JWEs share a common "alg" value space. The values used by the two specifications MUST be distinct, as the "alg" value MAY be used to determine whether the object is a JWS or JWE.

Likewise, additional reserved header parameter names MAY be defined via the IANA JSON Web Signature and Encryption Header Parameters registry [JWS]. 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. Cryptographic Algorithms for JWE

JWE uses cryptographic algorithms to encrypt the Content Master Key (CMK) and the Plaintext. This section specifies a set of specific algorithms for these purposes.

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 CMK, producing the JWE Encrypted Key, or to use key agreement to agree upon the CMK.

alg Parameter Value	Key Encryption or Agreement Algorithm	Implementation Requirements
RSA1_5 RSA-OAEP	RSAES-PKCS1-V1_5 [RFC3447] RSAES using Optimal Asymmetric Encryption Padding (OAEP) [RFC3447], with the default parameters specified by RFC 3447 in Section A.2.1	REQUIRED OPTIONAL
A128KW	Advanced Encryption Standard (AES) Key Wrap Algorithm [RFC3394] using 128 bit keys	RECOMMENDED
A256KW	AES Key Wrap Algorithm using 256 bit keys	RECOMMENDED
dir	Direct use of a shared symmetric key as the Content Master Key (CMK) for the block encryption step (rather than using the symmetric key to wrap the CMK)	RECOMMENDED
ECDH-ES	Elliptic Curve Diffie-Hellman Ephemeral Static [RFC6090] key agreement using the Concat KDF, as defined in Section 5.8.1 of [NIST.800-56A], with the agreed-upon key being used directly as the Content Master Key (CMK) (rather than being used to wrap the CMK), as specified in Section 4.7	RECOMMENDED+
ECDH-ES+A128KW	Elliptic Curve Diffie-Hellman Ephemeral Static key agreement per "ECDH-ES" and Section 4.7, but where the agreed-upon key is used to wrap the Content Master Key (CMK) with the "A128KW" function (rather than being used directly as the CMK)	RECOMMENDED
ECDH-ES+A256KW	Elliptic Curve Diffie-Hellman Ephemeral Static key agreement per "ECDH-ES" and Section 4.7, but where the agreed-upon key is used to wrap the Content Master Key (CMK) with the "A256KW" function (rather than being used directly as the CMK)	RECOMMENDED

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

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

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

enc Parameter Value	Block Encryption Algorithm	Implementation Requirements
A128CBC+HS256	Composite AEAD algorithm using Advanced Encryption Standard (AES) in Cipher Block Chaining (CBC) mode with PKCS #5 padding [AES] [NIST.800-38A] with an integrity calculation using HMAC SHA-256, using a 256 bit CMK (and 128 bit CEK) as specified in Section 4.8	REQUIRED
A256CBC+HS512	Composite AEAD algorithm using AES in CBC mode with PKCS #5 padding with an integrity calculation using HMAC SHA-512, using a 512 bit CMK (and 256 bit CEK) as specified in Section 4.8	REQUIRED
A128GCM	AES in Galois/Counter Mode (GCM) [AES] [NIST.800-38D] using 128 bit keys	RECOMMENDED
A256GCM	AES GCM using 256 bit keys	RECOMMENDED

All the names are short because a core goal of JWE is for the representations to be compact. However, there is no a priori length restriction on "alg" values.

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

4.3. Key Encryption with RSAES-PKCS1-V1_5

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

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

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

4.4. Key Encryption with RSAES OAEP

This section defines the specifics of encrypting a JWE CMK with RSAES using Optimal Asymmetric Encryption Padding (OAEP) [RFC3447], with the default parameters specified by RFC 3447 in Section A.2.1. The "alg" header parameter value "RSA-OAEP" is used in this case.

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

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

4.5. Key Encryption with AES Key Wrap

This section defines the specifics of encrypting a JWE CMK with the Advanced Encryption Standard (AES) Key Wrap Algorithm [RFC3394] using 128 or 256 bit keys. The "alg" header parameter values "A128KW" or "A256KW" are used in this case.

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

4.6. Direct Encryption with a Shared Symmetric Key

This section defines the specifics of directly performing symmetric key encryption without performing a key wrapping step. In this case, the shared symmetric key is used directly as the Content Master Key (CMK) value for the "enc" algorithm. An empty byte array is used as the JWE Encrypted Key value. The "alg" header parameter value "dir" is used in this case.

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

This section defines the specifics of key agreement with Elliptic Curve Diffie-Hellman Ephemeral Static [RFC6090], and using 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 Master Key (CMK) for the "enc" algorithm, or (2) as a symmetric key used to wrap the CMK with either the "A128KW" or

"A256KW" algorithms. The "alg" header parameter values "ECDH-ES", "ECDH-ES+A128KW", and "ECDH-ES+A256KW" are respectively used in this case.

In the direct case, the output of the Concat KDF MUST be a key of the same length as that used by the "enc" algorithm; in this case, the empty byte array is used as the JWE Encrypted Key value. In the key wrap case, the output of the Concat KDF MUST be a key of the length needed for the specified key wrap algorithm, either 128 or 256 bits respectively.

A new "epk" (ephemeral public key) value MUST be generated for each key agreement transaction.

4.7.1. Key Derivation for "ECDH-ES"

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 a byte array.

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

AlgorithmID This is set to the concatenation of keydatalen represented as a 32 bit big endian integer and the bytes of the UTF-8 representation of the "alg" header parameter value.

PartyUInfo If an "apu" (agreement PartyUInfo) header parameter is present, this is set to the result of base64url decoding the "apu" value; otherwise, it is set to the empty byte string.

PartyVInfo If an "apv" (agreement PartyVInfo) header parameter is present, this is set to the result of base64url decoding the "apv" value; otherwise, it is set to the empty byte string.

SuppPubInfo This is set to the empty byte string.

SuppPrivInfo This is set to the empty byte string.

For all three "alg" values, the digest function used is SHA-256.

4.8. Composite Plaintext Encryption Algorithms "A128CBC+HS256" and "A256CBC+HS512"

This section defines two composite "enc" algorithms that perform plaintext encryption using non-AEAD algorithms and add an integrity check calculation, so that the resulting composite algorithms are AEAD. These composite algorithms derive a Content Encryption Key (CEK) and a Content Integrity Key (CIK) from a Content Master Key, per Section 4.8.1. They perform block encryption with AES CBC, per Section 4.8.2. Finally, they add an integrity check using HMAC SHA-2 algorithms of matching strength, per Section 4.8.3.

A 256 bit Content Master Key (CMK) value is used with the "A128CBC+HS256" algorithm. A 512 bit Content Master Key (CMK) value is used with the "A256CBC+HS512" algorithm.

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

4.8.1. Key Derivation for "A128CBC+HS256" and "A256CBC+HS512"

The key derivation process derives CEK and CIK values from the CMK. This section defines the specifics of deriving keys for the "enc" algorithms "A128CBC+HS256" and "A256CBC+HS512".

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 or SHA-512, respectively. The Concat KDF parameters are set as follows:

Z This is set to the Content Master Key (CMK).

keydatalen This is set to the number of bits in the desired output key.

AlgorithmID This is set to the concatenation of keydatalen represented as a 32 bit big endian integer and the bytes of the UTF-8 representation of the "enc" header parameter value.

PartyUInfo If an "epu" (encryption PartyUInfo) header parameter is present, this is set to the result of base64url decoding the "epu" value; otherwise, it is set to the empty byte string.

PartyVInfo If an "epv" (encryption PartyVInfo) header parameter is present, this is set to the result of base64url decoding the "epv" value; otherwise, it is set to the empty byte string.

SuppPubInfo This is set to the bytes of one of the ASCII strings "Encryption" ([69, 110, 99, 114, 121, 112, 116, 105, 111, 110]) or "Integrity" ([73, 110, 116, 101, 103, 114, 105, 116, 121]) respectively, depending upon whether the CEK or CIK is being generated.

SuppPrivInfo This is set to the empty byte string.

To compute the CEK from the CMK, the ASCII label "Encryption" is used for the SuppPubInfo value. For "A128CBC+HS256", the keydatalen is 128 and the digest function used is SHA-256. For "A256CBC+HS512", the keydatalen is 256 and the digest function used is SHA-512.

To compute the CIK from the CMK, the ASCII label "Integrity" is used for the SuppPubInfo value. For "A128CBC+HS256", the keydatalen is 256 and the digest function used is SHA-256. For "A256CBC+HS512", the keydatalen is 512 and the digest function used is SHA-512.

Example derivation computations are shown in Appendices A.4 and A.5 of [JWE].

4.8.2. Encryption Calculation for "A128CBC+HS256" and "A256CBC+HS512"

This section defines the specifics of encrypting the JWE Plaintext with Advanced Encryption Standard (AES) in Cipher Block Chaining (CBC) mode with PKCS #5 padding [AES] [NIST.800-38A] using 128 or 256 bit keys. The "enc" header parameter values "A128CBC+HS256" or "A256CBC+HS512" are respectively used in this case.

The CEK is used as the encryption key.

Use of an initialization vector of size 128 bits is REQUIRED with these algorithms.

4.8.3. Integrity Calculation for "A128CBC+HS256" and "A256CBC+HS512"

This section defines the specifics of computing the JWE Integrity Value for the "enc" algorithms "A128CBC+HS256" and "A256CBC+HS512". This value is computed as a MAC of the JWE parameters to be secured.

The MAC input value is the bytes of the ASCII representation of the concatenation of the Encoded JWE Header, a period ('.') character, the Encoded JWE Encrypted Key, a second period character ('.'), the Encoded JWE Initialization Vector, a third period ('.') character,

and the Encoded JWE Ciphertext.

The CIK is used as the MAC key.

For "A128CBC+HS256", HMAC SHA-256 is used as the MAC algorithm. For "A256CBC+HS512", HMAC SHA-512 is used as the MAC algorithm.

The resulting MAC value is used as the JWE Integrity Value. The same integrity calculation is performed during decryption. During decryption, the computed integrity value must match the received JWE Integrity Value.

4.9. Plaintext Encryption with AES GCM

This section defines the specifics of encrypting the JWE Plaintext with Advanced Encryption Standard (AES) in Galois/Counter Mode (GCM) [AES] [NIST.800-38D] using 128 or 256 bit keys. The "enc" header parameter values "A128GCM" or "A256GCM" are used in this case.

The CMK is used as the encryption key.

Use of an initialization vector of size 96 bits is REQUIRED with this algorithm.

The "additional authenticated data" parameter is used to secure the header and key values. The "additional authenticated data" value used is the bytes of the ASCII representation of the concatenation of the Encoded JWE Header, a period ('.') character, the Encoded JWE Encrypted Key, a second period character ('.'), and the Encoded JWE Initialization Vector. This same "additional authenticated data" value is used when decrypting as well.

The requested size of the "authentication tag" output MUST be 128 bits, regardless of the key size.

As GCM is an AEAD algorithm, the JWE Integrity Value is set to be the "authentication tag" value produced by the encryption. During decryption, the received JWE Integrity Value is used as the "authentication tag" value.

Examples using this algorithm are shown in Appendices A.1 and A.3 of [JWE].

4.10. Additional Encryption Algorithms and Parameters

Additional algorithms MAY be used to protect JWEs with corresponding "alg" (algorithm) and "enc" (encryption method) header parameter values being defined to refer to them. New "alg" and "enc" header

parameter values SHOULD either be registered in the IANA JSON Web Signature and Encryption Algorithms registry Section 6.1 or be a URI that contains a Collision Resistant Namespace. In particular, it is permissible to use the algorithm identifiers defined in XML Encryption [W3C.REC-xmlenc-core-20021210], XML Encryption 1.1 [W3C.CR-xmlenc-core1-20120313], and related specifications as "alg" and "enc" values.

As indicated by the common registry, JWSs and JWEs share a common "alg" value space. The values used by the two specifications MUST be distinct, as the "alg" value MAY be used to determine whether the object is a JWS or JWE.

Likewise, additional reserved header parameter names MAY be defined via the IANA JSON Web Signature and Encryption Header Parameters registry [JWS]. 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.

5. Cryptographic Algorithms for JWK

A JSON Web Key (JWK) [JWK] is a JavaScript Object Notation (JSON) [RFC4627] data structure that represents a public key. A JSON Web Key Set (JWK Set) is a JSON data structure for representing a set of JWKs. This section specifies a set of algorithm families to be used for those public keys and the algorithm family specific parameters for representing those keys.

5.1. "alg" (Algorithm Family) Parameter Values for JWK

The table below is the set of "alg" (algorithm family) parameter values that are defined by this specification for use in JWKs.

alg Parameter Value	Algorithm Family	Implementation Requirements
EC	Elliptic Curve [DSS] key family	RECOMMENDED+
RSA	RSA [RFC3447] key family	REQUIRED

All the names are short because a core goal of JWK is for the representations to be compact. However, there is no a priori length restriction on "alg" values.

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

5.2. JWK Parameters for Elliptic Curve Keys

JWKs can represent Elliptic Curve [DSS] keys. In this case, the "alg" member value MUST be "EC". Furthermore, these additional members MUST be present:

5.2.1. "crv" (Curve) Parameter

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

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

Additional "crv" values MAY be used, provided they are understood by implementations using that Elliptic Curve key. The "crv" value is a case sensitive string.

5.2.2. "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 coordinate's big endian representation as a byte array. The array representation MUST not be shortened to omit any leading zero bytes contained in the value. For instance, when representing 521 bit integers, the byte array to be base64url encoded MUST contain 66 bytes, including any leading zero bytes.

5.2.3. "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 coordinate's big endian representation as a byte array. The array representation MUST not be shortened to omit any leading zero bytes contained in the value. For instance, when representing 521 bit integers, the byte array to be base64url encoded MUST contain 66 bytes, including any leading zero bytes.

5.3. JWK Parameters for RSA Keys

JWKs can represent RSA [RFC3447] keys. In this case, the "alg" member value MUST be "RSA". Furthermore, these additional members MUST be present:

5.3.1. "mod" (Modulus) Parameter

The "mod" (modulus) member contains the modulus value for the RSA public key. It is represented as the base64url encoding of the value's unsigned big endian representation as a byte array. The array representation MUST not be shortened to omit any leading zero bytes. For instance, when representing 2048 bit integers, the byte array to be base64url encoded MUST contain 256 bytes, including any leading zero bytes.

5.3.2. "xpo" (Exponent) Parameter

The "xpo" (exponent) member contains the exponent value for the RSA public key. It is represented as the base64url encoding of the value's unsigned big endian representation as a byte array. The array representation MUST utilize the minimum number of bytes to represent the value. For instance, when representing the value 65537, the byte array to be base64url encoded MUST consist of the three bytes [1, 0, 1].

5.4. Additional Key Algorithm Families and Parameters

Public keys using additional algorithm families MAY be represented using JWK data structures with corresponding "alg" (algorithm family) parameter values being defined to refer to them. New "alg" parameter values SHOULD either be registered in the IANA JSON Web Key Algorithm Families registry Section 6.2 or be a URI that contains a Collision Resistant Namespace.

Likewise, parameters for representing keys for additional algorithm families or additional key properties SHOULD either be registered in the IANA JSON Web Key Parameters registry [JWK] or be a URI that contains a Collision Resistant Namespace.

6. IANA Considerations

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

Values are registered with a Specification Required [RFC5226] after a two-week review period on the [TBD]@ietf.org mailing list, on the

advice of one or more Designated Experts. However, to allow for the allocation of values prior to publication, the Designated Expert(s) may approve registration once they are satisfied that such a specification will be published.

Registration requests must be sent to the [TBD]@ietf.org mailing list for review and comment, with an appropriate subject (e.g., "Request for access token type: example"). [[Note to RFC-EDITOR: The name of the mailing list should be determined in consultation with the IESG and IANA. Suggested name: jose-reg-review.]]

Within the review period, the Designated Expert(s) will either approve or deny the registration request, communicating this decision to the review list and IANA. Denials should include an explanation and, if applicable, suggestions as to how to make the request successful.

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

6.1. JSON Web Signature and Encryption Algorithms Registry

This specification establishes the IANA JSON Web Signature and Encryption Algorithms registry for values of the JWS and JWE "alg" (algorithm) and "enc" (encryption method) header parameters. The registry records the algorithm name, the algorithm usage locations from the set "alg" and "enc", implementation requirements, and a reference to the specification that defines it. The same algorithm name may be registered multiple times, provided that the sets of usage locations are disjoint. The implementation requirements of an algorithm may be changed over time by the Designated Experts(s) as the cryptographic landscape evolves, for instance, to change the status of an algorithm to DEPRECATED, or to change the status of an algorithm from OPTIONAL to RECOMMENDED or REQUIRED.

6.1.1. Registration Template

Algorithm Name:

The name requested (e.g., "example"). This name is case sensitive. Names that match other registered names in a case insensitive manner SHOULD NOT be accepted.

Algorithm Usage Location(s):

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

Implementation Requirements:

The algorithm implementation requirements, which must be one the words REQUIRED, RECOMMENDED, OPTIONAL, or DEPRECATED. Optionally, the word may 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 "IETF". For others, give the name of the responsible party. Other details (e.g., postal address, email address, home page URI) may also be included.

Specification Document(s):

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

6.1.2. Initial Registry Contents

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

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

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

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

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

- o Algorithm Name: "RS512"
- o Algorithm Usage Location(s): "alg"
- o Implementation Requirements: OPTIONAL
- o Change Controller: IETF
- o Specification Document(s): Section 3.1 of [[this document]]

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

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

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

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

- o Algorithm Name: "RSA1_5"
- o Algorithm Usage Location(s): "alg"
- o Implementation Requirements: REQUIRED
- o Change Controller: IETF
- o Specification Document(s): Section 4.1 of [[this document]]

- o Algorithm Name: "RSA-OAEP"
- o Algorithm Usage Location(s): "alg"
- o Implementation Requirements: OPTIONAL
- o Change Controller: IETF

- o Specification Document(s): Section 4.1 of [[this document]]
- o Algorithm Name: "A128KW"
- o Algorithm Usage Location(s): "alg"
- o Implementation Requirements: RECOMMENDED
- o Change Controller: IETF
- o Specification Document(s): Section 4.1 of [[this document]]
- o Algorithm Name: "A256KW"
- o Algorithm Usage Location(s): "alg"
- o Implementation Requirements: RECOMMENDED
- o Change Controller: IETF
- o Specification Document(s): Section 4.1 of [[this document]]
- o Algorithm Name: "dir"
- o Algorithm Usage Location(s): "alg"
- o Implementation Requirements: RECOMMENDED
- o Change Controller: IETF
- o Specification Document(s): Section 4.1 of [[this document]]
- o Algorithm Name: "ECDH-ES"
- o Algorithm Usage Location(s): "alg"
- o Implementation Requirements: RECOMMENDED+
- o Change Controller: IETF
- o Specification Document(s): Section 4.1 of [[this document]]
- o Algorithm Name: "ECDH-ES+A128KW"
- o Algorithm Usage Location(s): "alg"
- o Implementation Requirements: RECOMMENDED
- o Change Controller: IETF
- o Specification Document(s): Section 4.1 of [[this document]]
- o Algorithm Name: "ECDH-ES+A256KW"
- o Algorithm Usage Location(s): "alg"
- o Implementation Requirements: RECOMMENDED
- o Change Controller: IETF
- o Specification Document(s): Section 4.1 of [[this document]]
- o Algorithm Name: "A128CBC+HS256"
- o Algorithm Usage Location(s): "enc"
- o Implementation Requirements: REQUIRED
- o Change Controller: IETF
- o Specification Document(s): Section 4.2 of [[this document]]
- o Algorithm Name: "A256CBC+HS512"
- o Algorithm Usage Location(s): "enc"

- o Implementation Requirements: REQUIRED
- o Change Controller: IETF
- o Specification Document(s): Section 4.2 of [[this document]]

- o Algorithm Name: "A128GCM"
- o Algorithm Usage Location(s): "enc"
- o Implementation Requirements: RECOMMENDED
- o Change Controller: IETF
- o Specification Document(s): Section 4.2 of [[this document]]

- o Algorithm Name: "A256GCM"
- o Algorithm Usage Location(s): "enc"
- o Implementation Requirements: RECOMMENDED
- o Change Controller: IETF
- o Specification Document(s): Section 4.2 of [[this document]]

6.2. JSON Web Key Algorithm Families Registry

This specification establishes the IANA JSON Web Key Algorithm Families registry for values of the JWK "alg" (algorithm family) parameter. The registry records the "alg" value and a reference to the specification that defines it. This specification registers the values defined in Section 5.1.

6.2.1. Registration Template

"alg" Parameter Value:

The name requested (e.g., "example"). This name is case sensitive. Names that match other registered names in a case insensitive manner SHOULD NOT be accepted.

Change Controller:

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

Implementation Requirements:

The algorithm implementation requirements, which must be one the words REQUIRED, RECOMMENDED, OPTIONAL, or DEPRECATED. Optionally, the word may be followed by a "+" or "-". The use of "+" indicates that the requirement strength is likely to be increased in a future version of the specification. The use of "-" indicates that the requirement strength is likely to be decreased in a future version of the specification.

Specification Document(s):

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

6.2.2. Initial Registry Contents

- o "alg" Parameter Value: "EC"
- o Implementation Requirements: RECOMMENDED+
- o Change Controller: IETF
- o Specification Document(s): Section 5.1 of [[this document]]

- o "alg" Parameter Value: "RSA"
- o Implementation Requirements: REQUIRED
- o Change Controller: IETF
- o Specification Document(s): Section 5.1 of [[this document]]

6.3. JSON Web Key Parameters Registration

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

6.3.1. Registry Contents

- o Parameter Name: "crv"
- o Change Controller: IETF
- o Specification Document(s): Section 5.2.1 of [[this document]]

- o Parameter Name: "x"
- o Change Controller: IETF
- o Specification Document(s): Section 5.2.2 of [[this document]]

- o Parameter Name: "y"
- o Change Controller: IETF
- o Specification Document(s): Section 5.2.3 of [[this document]]

- o Parameter Name: "mod"
- o Change Controller: IETF
- o Specification Document(s): Section 5.3.1 of [[this document]]

- o Parameter Name: "xpo"
- o Change Controller: IETF
- o Specification Document(s): Section 5.3.2 of [[this document]]

7. Security Considerations

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

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

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

Algorithms of matching strength 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.

While Section 8 of RFC 3447 [RFC3447] explicitly calls for people not to adopt RSASSA-PKCS1 for new applications and instead requests that people transition to RSASSA-PSS, this specification does include RSASSA-PKCS1, for interoperability reasons, because it commonly implemented.

Keys used with RSAES-PKCS1-v1_5 must follow the constraints in Section 7.2 of RFC 3447 [RFC3447]. In particular, keys with a low public key exponent value must not be used.

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

Receiving agents that validate signatures and sending agents that encrypt messages need to be cautious of cryptographic processing usage when validating signatures and encrypting messages using keys larger than those mandated in this specification. An attacker could send certificates with keys that would result in excessive cryptographic processing, for example, keys larger than those mandated in this specification, which could swamp the processing element. Agents that use such keys without first validating the

certificate to a trust anchor are advised to have some sort of cryptographic resource management system to prevent such attacks.

8. References

8.1. Normative References

- [AES] National Institute of Standards and Technology (NIST), "Advanced Encryption Standard (AES)", FIPS PUB 197, November 2001.
- [DSS] National Institute of Standards and Technology, "Digital Signature Standard (DSS)", FIPS PUB 186-3, June 2009.
- [JWE] Jones, M., Rescorla, E., and J. Hildebrand, "JSON Web Encryption (JWE)", October 2012.
- [JWK] Jones, M., "JSON Web Key (JWK)", October 2012.
- [JWS] Jones, M., Bradley, J., and N. Sakimura, "JSON Web Signature (JWS)", October 2012.
- [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 (Revised)", NIST PUB 800-56A, March 2007.
- [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.
- [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.
- [RFC4627] Crockford, D., "The application/json Media Type for JavaScript Object Notation (JSON)", RFC 4627, July 2006.
- [RFC4648] Josefsson, S., "The Base16, Base32, and Base64 Data Encodings", RFC 4648, October 2006.
- [RFC5116] McGrew, D., "An Interface and Algorithms for Authenticated Encryption", RFC 5116, January 2008.
- [RFC5226] Narten, T. and H. Alvestrand, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 5226, May 2008.
- [RFC6090] McGrew, D., Igoe, K., and M. Salter, "Fundamental Elliptic Curve Cryptography Algorithms", RFC 6090, February 2011.
- [SHS] National Institute of Standards and Technology, "Secure Hash Standard (SHS)", FIPS PUB 180-3, October 2008.
- [USASCII] American National Standards Institute, "Coded Character Set -- 7-bit American Standard Code for Information Interchange", ANSI X3.4, 1986.

8.2. Informative References

- [CanvasApp] Facebook, "Canvas Applications", 2010.
- [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", 2011.
- [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.

[RFC3275] Eastlake, D., Reagle, J., and D. Solo, "(Extensible Markup Language) XML-Signature Syntax and Processing", RFC 3275, March 2002.

[RFC4122] Leach, P., Mealling, M., and R. Salz, "A Universally Unique Identifier (UUID) URN Namespace", RFC 4122, July 2005.

[W3C.CR-xmlsig-core2-20120124]

Roessler, T., Yiu, K., Solo, D., Reagle, J., Datta, P., Eastlake, D., Hirsch, F., and S. Cantor, "XML Signature Syntax and Processing Version 2.0", World Wide Web Consortium CR CR-xmlsig-core2-20120124, January 2012, <<http://www.w3.org/TR/2012/CR-xmlsig-core2-20120124>>.

[W3C.CR-xmlenc-core1-20120313]

Eastlake, D., Reagle, J., Hirsch, F., and T. Roessler, "XML Encryption Syntax and Processing Version 1.1", World Wide Web Consortium CR CR-xmlenc-core1-20120313, March 2012, <<http://www.w3.org/TR/2012/CR-xmlenc-core1-20120313>>.

[W3C.REC-xmlenc-core-20021210]

Eastlake, D. and J. Reagle, "XML Encryption Syntax and Processing", World Wide Web Consortium Recommendation REC-xmlenc-core-20021210, December 2002, <<http://www.w3.org/TR/2002/REC-xmlenc-core-20021210>>.

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

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

Algor ithm	JWS	XML DSIG	JCA	OID
HMAC using SHA-2 56 hash algo rithm	HS2 56	http://www.w3.org/2001/04/ xmlsig-more#hmac-sha256	HmacSHA2 56	1.2.840.113 549.2.9
HMAC using SHA-3 84 hash algo rithm	HS3 84	http://www.w3.org/2001/04/ xmlsig-more#hmac-sha384	HmacSHA3 84	1.2.840.113 549.2.10
HMAC using SHA-5 12 hash algo rithm	HS5 12	http://www.w3.org/2001/04/ xmlsig-more#hmac-sha512	HmacSHA5 12	1.2.840.113 549.2.11
RSASS A usin gSHA- 256 has h alg orith m	RS2 56	http://www.w3.org/2001/04/ xmlsig-more#rsa-sha256	SHA256wi thRSA	1.2.840.113 549.1.1.11
RSASS A usin gSHA- 384 has h alg orith m	RS3 84	http://www.w3.org/2001/04/ xmlsig-more#rsa-sha384	SHA384wi thRSA	1.2.840.113 549.1.1.12

RSASSA	RS512	http://www.w3.org/2001/04/xmldsig-more#rsa-sha512	SHA512withRSA	1.2.840.113549.1.1.13
using SHA-512 hash algorithm				
ECDSA using P-256 curve and SHA-256 hash algorithm	ES256	http://www.w3.org/2001/04/xmldsig-more#ecdsa-sha256	SHA256withECDSA	1.2.840.10045.4.3.2
ECDSA using P-384 curve and SHA-384 hash algorithm	ES384	http://www.w3.org/2001/04/xmldsig-more#ecdsa-sha384	SHA384withECDSA	1.2.840.10045.4.3.3
ECDSA using P-521 curve and SHA-512 hash algorithm	ES512	http://www.w3.org/2001/04/xmldsig-more#ecdsa-sha512	SHA512withECDSA	1.2.840.10045.4.3.4

Appendix B. Encryption Algorithm Identifier Cross-Reference

This appendix contains a table cross-referencing the "alg" (algorithm) and "enc" (encryption method) values used in this specification with the equivalent identifiers used by other standards and software packages. See XML Encryption

[W3C.REC-xmlenc-core-20021210], XML Encryption 1.1 [W3C.CR-xmlenc-core1-20120313], and Java Cryptography Architecture [JCA] for more information about the names defined by those documents.

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

Algorithm	JWE	XML ENC	JCA
RSAPKCS1-v1_5	RSA1_5	http://www.w3.org/2001/04/xmlenc#rsa-1_5	RSA/ECB/PKCS1Padding
RSAPKCS1-v1_5 using Optimal Asymmetric Encryption Padding (OAEP)	RSA-OAEP	http://www.w3.org/2001/04/xmlenc#rsa-oaep-mgf1p	RSA/ECB/OAEPWithSHA-1AndMGF1Padding
Elliptic Curve Diffie-Hellman EphemeralStatic Advanced Encryption Standard(AES) Key Wrap Algorithm using 128 bit keys	ECDH-ES	http://www.w3.org/2009/xmlenc11#ECDH-ES	
AES Key Wrap Algorithm using 128 bit keys	A128KW	http://www.w3.org/2001/04/xmlenc#kw-aes128	
AES Key Wrap Algorithm using 256 bit keys	A256KW	http://www.w3.org/2001/04/xmlenc#kw-aes256	

AES in Cipher Block Chaining (CBC) mode with PKCS #5 padding using 128 bit keys	A128CBC+HS256	http://www.w3.org/2001/04/xmlenc#aes128-cbc	AES/CBC/PKCS5Padding
AES in CBC mode with PKCS #5 padding using 256 bit keys	A256CBC+HS512	http://www.w3.org/2001/04/xmlenc#aes256-cbc	AES/CBC/PKCS5Padding
AES in Galois/Counter Mode (GCM) using 128 bit keys	A128GCM	http://www.w3.org/2009/xmlenc11#aes128-gcm	AES/GCM/NoPadding
AES GCM using 256 bit keys	A256GCM	http://www.w3.org/2009/xmlenc11#aes256-gcm	AES/GCM/NoPadding

Appendix C. 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. Dirk Balfanz, John Bradley, Yaron Y. Golan, John Panzer, Nat Sakimura, and Paul Tarjan all made significant contributions to the design of this specification and its related specifications.

Jim Schaad and Karen O'Donoghue chaired the JOSE working group and Sean Turner and Stephen Farrell served as Security area directors

during the creation of this specification.

Appendix D. Open Issues

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

The following items remain to be considered or done in this draft:

- o No known open issues.

Appendix E. Document History

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

-06

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

-05

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

- o Updated open issues.

-04

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

-03

- o Always use a 128 bit "authentication tag" size for AES GCM, regardless of the key size.
- o Specified that use of a 128 bit IV is REQUIRED with AES CBC. It was previously RECOMMENDED.
- o Removed key size language for ECDSA algorithms, since the key size is implied by the algorithm being used.
- o Stated that the "int" key size must be the same as the hash output size (and not larger, as was previously allowed) so that its size is defined for key generation purposes.
- o Added the "kdf" (key derivation function) header parameter to provide crypto agility for key derivation. The default KDF remains the Concat KDF with the SHA-256 digest function.
- o Clarified that the "mod" and "exp" values are unsigned.
- o Added Implementation Requirements columns to algorithm tables and Implementation Requirements entries to algorithm registries.
- o Changed AES Key Wrap to RECOMMENDED.
- o Moved registries JSON Web Signature and Encryption Header Parameters and JSON Web Signature and Encryption Type Values to the JWS specification.
- o Moved JSON Web Key Parameters registry to the JWK specification.

- o Changed registration requirements from RFC Required to Specification Required with Expert Review.
- o Added Registration Template sections for defined registries.
- o Added Registry Contents sections to populate registry values.
- o No longer say "the UTF-8 representation of the JWS Secured Input (which is the same as the ASCII representation)". Just call it "the ASCII representation of the JWS Secured Input".
- o Added "Collision Resistant Namespace" to the terminology section.
- o Numerous editorial improvements.

-02

- o For AES GCM, use the "additional authenticated data" parameter to provide integrity for the header, encrypted key, and ciphertext and use the resulting "authentication tag" value as the JWE Integrity Value.
- o Defined minimum required key sizes for algorithms without specified key sizes.
- o Defined KDF output key sizes.
- o Specified the use of PKCS #5 padding with AES CBC.
- o Generalized text to allow key agreement to be employed as an alternative to key wrapping or key encryption.
- o Clarified that ECDH-ES is a key agreement algorithm.
- o Required implementation of AES-128-KW and AES-256-KW.
- o Removed the use of "A128GCM" and "A256GCM" for key wrapping.
- o Removed "A512KW" since it turns out that it's not a standard algorithm.
- o Clarified the relationship between "typ" header parameter values and MIME types.
- o Generalized language to refer to Message Authentication Codes (MACs) rather than Hash-based Message Authentication Codes (HMACs) unless in a context specific to HMAC algorithms.

- o Established registries: JSON Web Signature and Encryption Header Parameters, JSON Web Signature and Encryption Algorithms, JSON Web Signature and Encryption "typ" Values, JSON Web Key Parameters, and JSON Web Key Algorithm Families.
- o Moved algorithm-specific definitions from JWK to JWA.
- o Reformatted to give each member definition its own section heading.

-01

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

-00

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

Author's Address

Michael B. Jones
Microsoft

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

JOSE Working Group
Internet-Draft
Intended status: Standards Track
Expires: April 18, 2013

M. Jones
Microsoft
E. Rescorla
RTFM
J. Hildebrand
Cisco
October 15, 2012

JSON Web Encryption (JWE)
draft-ietf-jose-json-web-encryption-06

Abstract

JSON Web Encryption (JWE) is a means of representing encrypted content using JavaScript Object Notation (JSON) data structures. Cryptographic algorithms and identifiers for use with this specification are described in the separate JSON Web Algorithms (JWA) specification. Related digital signature and MAC capabilities are described in the separate JSON Web Signature (JWS) specification.

Status of this Memo

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

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

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

This Internet-Draft will expire on April 18, 2013.

Copyright Notice

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

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

to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction	5
1.1. Notational Conventions	5
2. Terminology	5
3. JSON Web Encryption (JWE) Overview	7
3.1. Example JWE with an Integrated Integrity Check	8
3.2. Example JWE with a Separate Integrity Check	9
4. JWE Header	11
4.1. Reserved Header Parameter Names	12
4.1.1. "alg" (Algorithm) Header Parameter	12
4.1.2. "enc" (Encryption Method) Header Parameter	12
4.1.3. "epk" (Ephemeral Public Key) Header Parameter	13
4.1.4. "zip" (Compression Algorithm) Header Parameter	13
4.1.5. "jku" (JWK Set URL) Header Parameter	13
4.1.6. "jwk" (JSON Web Key) Header Parameter	13
4.1.7. "x5u" (X.509 URL) Header Parameter	13
4.1.8. "x5t" (X.509 Certificate Thumbprint) Header Parameter	14
4.1.9. "x5c" (X.509 Certificate Chain) Header Parameter	14
4.1.10. "kid" (Key ID) Header Parameter	14
4.1.11. "typ" (Type) Header Parameter	15
4.1.12. "cty" (Content Type) Header Parameter	15
4.1.13. "apu" (Agreement PartyUInfo) Header Parameter	15
4.1.14. "apv" (Agreement PartyVInfo) Header Parameter	15
4.1.15. "epu" (Encryption PartyUInfo) Header Parameter	15
4.1.16. "epv" (Encryption PartyVInfo) Header Parameter	16
4.2. Public Header Parameter Names	16
4.3. Private Header Parameter Names	16
5. Message Encryption	16
6. Message Decryption	18
7. CMK Encryption	19
8. Encrypting JWEs with Cryptographic Algorithms	19
9. IANA Considerations	20
9.1. Registration of JWE Header Parameter Names	20
9.1.1. Registry Contents	20
9.2. JSON Web Signature and Encryption Type Values Registration	21
9.2.1. Registry Contents	21
9.3. Media Type Registration	21
9.3.1. Registry Contents	21
10. Security Considerations	22

11. References	22
11.1. Normative References	22
11.2. Informative References	24
Appendix A. JWE Examples	24
A.1. Example JWE using RSAES OAEP and AES GCM	24
A.1.1. JWE Header	25
A.1.2. Encoded JWE Header	25
A.1.3. Content Master Key (CMK)	25
A.1.4. Key Encryption	25
A.1.5. Encoded JWE Encrypted Key	28
A.1.6. Initialization Vector	28
A.1.7. "Additional Authenticated Data" Parameter	28
A.1.8. Plaintext Encryption	29
A.1.9. Encoded JWE Ciphertext	29
A.1.10. Encoded JWE Integrity Value	30
A.1.11. Complete Representation	30
A.1.12. Validation	30
A.2. Example JWE using RSAES-PKCS1-V1_5 and AES CBC	30
A.2.1. JWE Header	31
A.2.2. Encoded JWE Header	31
A.2.3. Content Master Key (CMK)	31
A.2.4. Key Encryption	31
A.2.5. Encoded JWE Encrypted Key	34
A.2.6. Key Derivation	34
A.2.7. Initialization Vector	34
A.2.8. Plaintext Encryption	34
A.2.9. Encoded JWE Ciphertext	35
A.2.10. Secured Input Value	35
A.2.11. JWE Integrity Value	36
A.2.12. Encoded JWE Integrity Value	36
A.2.13. Complete Representation	36
A.2.14. Validation	37
A.3. Example JWE using AES Key Wrap and AES GCM	37
A.3.1. JWE Header	37
A.3.2. Encoded JWE Header	38
A.3.3. Content Master Key (CMK)	38
A.3.4. Key Encryption	38
A.3.5. Encoded JWE Encrypted Key	38
A.3.6. Initialization Vector	38
A.3.7. "Additional Authenticated Data" Parameter	39
A.3.8. Plaintext Encryption	39
A.3.9. Encoded JWE Ciphertext	39
A.3.10. Encoded JWE Integrity Value	40
A.3.11. Complete Representation	40
A.3.12. Validation	40
A.4. Example Key Derivation for "enc" value "A128CBC+HS256"	40
A.4.1. CEK Generation	41
A.4.2. CIK Generation	42

A.5. Example Key Derivation for "enc" value "A256CBC+HS512" . .	42
A.5.1. CEK Generation	43
A.5.2. CIK Generation	44
Appendix B. Acknowledgements	45
Appendix C. Open Issues	45
Appendix D. Document History	46
Authors' Addresses	49

1. Introduction

JSON Web Encryption (JWE) is a compact encryption format intended for space constrained environments such as HTTP Authorization headers and URI query parameters. It represents this content using JavaScript Object Notation (JSON) [RFC4627] based data structures. The JWE cryptographic mechanisms encrypt and provide integrity protection for arbitrary sequences of bytes.

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

1.1. Notational Conventions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in Key words for use in RFCs to Indicate Requirement Levels [RFC2119].

2. Terminology

JSON Web Encryption (JWE) A data structure representing an encrypted message. The structure consists of five parts: the JWE Header, the JWE Encrypted Key, the JWE Initialization Vector, the JWE Ciphertext, and the JWE Integrity Value.

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

Ciphertext An encrypted representation of the Plaintext.

Content Encryption Key (CEK) A symmetric key used to encrypt the Plaintext for the recipient to produce the Ciphertext.

Content Integrity Key (CIK) A key used with a MAC function to ensure the integrity of the Ciphertext and the parameters used to create it.

Content Master Key (CMK) A key from which the CEK and CIK are derived. When key wrapping or key encryption are employed, the CMK is randomly generated and encrypted to the recipient as the JWE Encrypted Key. When direct encryption with a shared symmetric key is employed, the CMK is the shared key. When key agreement without key wrapping is employed, the CMK is the result of the key

agreement algorithm.

JWE Header A string representing a JSON object that describes the encryption operations applied to create the JWE Encrypted Key, the JWE Ciphertext, and the JWE Integrity Value.

JWE Encrypted Key When key wrapping or key encryption are employed, the Content Master Key (CMK) is encrypted with the intended recipient's key and the resulting encrypted content is recorded as a byte array, which is referred to as the JWE Encrypted Key. Otherwise, when direct encryption with a shared or agreed upon symmetric key is employed, the JWE Encrypted Key is the empty byte array.

JWE Initialization Vector A byte array containing the Initialization Vector used when encrypting the Plaintext.

JWE Ciphertext A byte array containing the Ciphertext.

JWE Integrity Value A byte array containing a MAC value that ensures the integrity of the Ciphertext and the parameters used to create it.

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

Encoded JWE Header Base64url encoding of the bytes of the UTF-8 [RFC3629] representation of the JWE Header.

Encoded JWE Encrypted Key Base64url encoding of the JWE Encrypted Key.

Encoded JWE Initialization Vector Base64url encoding of the JWE Initialization Vector.

Encoded JWE Ciphertext Base64url encoding of the JWE Ciphertext.

Encoded JWE Integrity Value Base64url encoding of the JWE Integrity Value.

Header Parameter Name The name of a member of the JSON object representing a JWE Header.

Header Parameter Value The value of a member of the JSON object representing a JWE Header.

JWE Compact Serialization A representation of the JWE as the concatenation of the Encoded JWE Header, the Encoded JWE Encrypted Key, the Encoded JWE Initialization Vector, the Encoded JWE Ciphertext, and the Encoded JWE Integrity Value in that order, with the five strings being separated by four period ('.') characters.

AEAD Algorithm An Authenticated Encryption with Associated Data (AEAD) [RFC5116] encryption algorithm is one that provides an integrated content integrity check. AES Galois/Counter Mode (GCM) is one such algorithm.

Collision Resistant Namespace A namespace that allows names to be allocated in a manner such that they are highly unlikely to collide with other names. For instance, collision resistance can be achieved through administrative delegation of portions of the namespace or through use of collision-resistant name allocation functions. Examples of Collision Resistant Namespaces include: Domain Names, Object Identifiers (OIDs) as defined in the ITU-T X.660 and X.670 Recommendation series, and Universally Unique Identifiers (UUIDs) [RFC4122]. When using an administratively delegated namespace, the definer of a name needs to take reasonable precautions to ensure they are in control of the portion of the namespace they use to define the name.

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

3. JSON Web Encryption (JWE) Overview

JWE represents encrypted content using JSON data structures and base64url encoding. The representation consists of five parts: the JWE Header, the JWE Encrypted Key, the JWE Initialization Vector, the JWE Ciphertext, and the JWE Integrity Value. In the Compact Serialization, the five parts are base64url-encoded for transmission, and represented as the concatenation of the encoded strings in that order, with the five strings being separated by four period ('.') characters. (A JSON Serialization for this information is defined in the separate JSON Web Encryption JSON Serialization (JWE-JS) [JWE-JS] specification.)

JWE utilizes encryption to ensure the confidentiality of the Plaintext. JWE adds a content integrity check if not provided by the underlying encryption algorithm.

3.1. Example JWE with an Integrated Integrity Check

This example encrypts the plaintext "Live long and prosper." to the recipient using RSAES OAEP and AES GCM. The AES GCM algorithm has an integrated integrity check.

The following example JWE Header declares that:

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

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

Base64url encoding the bytes of the UTF-8 representation of the JWE Header yields this Encoded JWE Header value:

```
eyJhbGciOiJSU0EtT0FFUCIsImVuYyI6IkeEYNTZHQ00ifQ
```

The remaining steps to finish creating this JWE are:

- o Generate a random Content Master Key (CMK)
- o Encrypt the CMK with the recipient's public key using the RSAES OAEP algorithm to produce the JWE Encrypted Key
- o Base64url encode the JWE Encrypted Key to produce the Encoded JWE Encrypted Key
- o Generate a random JWE Initialization Vector
- o Base64url encode the JWE Initialization Vector to produce the Encoded JWE Initialization Vector
- o Concatenate the Encoded JWE Header value, a period character ('.'), the Encoded JWE Encrypted Key, a second period character ('.'), and the Encoded JWE Initialization Vector to create the "additional authenticated data" parameter for the AES GCM algorithm

- o Encrypt the Plaintext with AES GCM, using the CMK as the encryption key, the JWE Initialization Vector, and the "additional authenticated data" value above, requesting a 128 bit "authentication tag" output
- o Base64url encode the resulting Ciphertext to create the Encoded JWE Ciphertext
- o Base64url encode the resulting "authentication tag" to create the Encoded JWE Integrity Value
- o Assemble the final representation: The Compact Serialization of this result is the concatenation of the Encoded JWE Header, the Encoded JWE Encrypted Key, the Encoded JWE Initialization Vector, the Encoded JWE Ciphertext, and the Encoded JWE Integrity Value in that order, with the five strings being separated by four period ('.') characters.

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

```
eyJhbGciOiJSU0EtT0FFUCIsImVuYyI6IkEyNTZHQ00ifQ.
M2XxbORKezKSzzQL_95-GjiudRBTqn_omS8z9xgoRb7L0Jw5UsEbxmtyHn2T7lm
rZLkjpg4Mp8gbhYoltPkeOHvAopz25-vZ8C2elcOaAo5WPcbSiufcB4DjBOM3t0UA
O6JHkWLuAEYoe58lclxIQneyKdaYSLbV9cKqoUoFQpvKWYRHZbfszIyfsal8rmgTj
zrtLDTpnc09DSJE24aQ8w3i8RXEDthW9TlJ6LsTH_vwHdwUgkI-tC2PNeGrnM-dN
SfzF3Y7-lwcGy0FsdXkPXyvtvDV7y4pZeeUiQ-0VdibIN2AjjfW60nfrPuOjepMFG
6BBBbR37pHcyzext9epOAQ.
48Vl_ALb6US04U3b.
_e2ltGGhac_peEFkLXr2dMPUziUkrw.
7V5ZDko0v_mf2Pac4JMiUg
```

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

3.2. Example JWE with a Separate Integrity Check

This example encrypts the plaintext "No matter where you go, there you are." to the recipient using RSAES-PKCS1-V1_5 and AES CBC. AES CBC does not have an integrated integrity check, so a separate integrity check calculation is performed using HMAC SHA-256, with separate encryption and integrity keys being derived from a master key using the Concat KDF with the SHA-256 digest function.

The following example JWE Header (with line breaks for display purposes only) declares that:

- o the Content Master Key is encrypted to the recipient using the RSAES-PKCS1-V1_5 algorithm to produce the JWE Encrypted Key and

- o the Plaintext is encrypted using the AES CBC algorithm with a 128 bit key to produce the Ciphertext, with the integrity of the Ciphertext and the parameters used to create it being secured using the HMAC SHA-256 algorithm.

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

Base64url encoding the bytes of the UTF-8 representation of the JWE Header yields this Encoded JWE Header value:

```
eyJhbGciOiJSU0ExXzUiLCJlbmMiOiJBMTI4Q0JDK0hTMjU2In0
```

The remaining steps to finish creating this JWE are like the previous example, but with an additional step to compute the separate integrity value:

- o Generate a random Content Master Key (CMK)
- o Encrypt the CMK with the recipient's public key using the RSAES-PKCS1-V1_5 algorithm to produce the JWE Encrypted Key
- o Base64url encode the JWE Encrypted Key to produce the Encoded JWE Encrypted Key
- o Generate a random JWE Initialization Vector
- o Base64url encode the JWE Initialization Vector to produce the Encoded JWE Initialization Vector
- o Use the Concat key derivation function to derive Content Encryption Key (CEK) and Content Integrity Key (CIK) values from the CMK
- o Encrypt the Plaintext with AES CBC using the CEK and JWE Initialization Vector to produce the Ciphertext
- o Base64url encode the resulting Ciphertext to create the Encoded JWE Ciphertext
- o Concatenate the Encoded JWE Header value, a period character ('.'), the Encoded JWE Encrypted Key, a second period character ('.'), the Encoded JWE Initialization Vector, a third period ('.') character, and the Encoded JWE Ciphertext to create the value to integrity protect
- o Compute the HMAC SHA-256 of this value using the CIK to create the JWE Integrity Value

- o Base64url encode the resulting JWE Integrity Value to create the Encoded JWE Integrity Value
- o Assemble the final representation: The Compact Serialization of this result is the concatenation of the Encoded JWE Header, the Encoded JWE Encrypted Key, the Encoded JWE Initialization Vector, the Encoded JWE Ciphertext, and the Encoded JWE Integrity Value in that order, with the five strings being separated by four period ('.') characters.

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

```
eyJhbGciOiJSU0ExXzUiLCJlbmMiOiJBMTI4Q0JDK0hTMjU2In0.
O6AqXqgVlJJ4c4lp5sXZd7bpGHAW6ARkHUeXQxDlcAW4-X1x0qtj_AN0mukqE0l4
Y6UOWJXIjY9-G1ELK-RQWrKH_StR-AM9H7GpKmSEji8QYOCMOjr-u9H1Lt_pBEie
G802SxWz0rbFTXRcj4BWLxcpCtjUZ3lAP-sc-L_eCZ5UNl0aSRNqFskuPkzRsFZR
DJqSSJeVOyJ7pZCQ83fli19Vgi_3R7XMUqluQuuc7ZHOWixi47jXlBTlWRZ5iFxa
S8G6J8wUrd4BKggAw3qX5XoIfXQVlQZE0Vmkq_zQSIo5LnFKyowooRcdsEuNh9B9
Mkyt0ZQE1G-jGdtHWjZSOA.
AxY8DcTdaGlsbGljb3RoZQ.
1eBWFgcrz40wC88cgV8rPgu3EfmClp4zT0kIxxfSF2zDJcQ-iEHkljQM95xAdr5Z.
RBGhYzE8_cZLHjJqqHuLhzbgWgL_wV3LDSUrcbkOiIA
```

See Appendix A.2 for the complete details of computing this JWE.

4. JWE Header

The members of the JSON object represented by the JWE Header describe the encryption applied to the Plaintext and optionally additional properties of the JWE. The Header Parameter Names within this object MUST be unique; JWEs with duplicate Header Parameter Names MUST be rejected. Implementations MUST understand the entire contents of the header; otherwise, the JWE MUST be rejected.

There are two ways of distinguishing whether a header is a JWS Header or a JWE Header. The first is by examining the "alg" (algorithm) header value. If the value represents a digital signature or MAC algorithm, or is the value "none", it is for a JWS; if it represents an encryption or key agreement algorithm, it is for a JWE. A second method is determining whether an "enc" (encryption method) member exists. If the "enc" member exists, it is a JWE; otherwise, it is a JWS. Both methods will yield the same result for all legal input values.

There are three classes of Header Parameter Names: Reserved Header Parameter Names, Public Header Parameter Names, and Private Header

Parameter Names.

4.1. Reserved Header Parameter Names

The following header parameter names are reserved with meanings as defined below. All the names are short because a core goal of JWE is for the representations to be compact.

Additional reserved header parameter names MAY be defined via the IANA JSON Web Signature and Encryption Header Parameters registry [JWS]. 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 encrypt or determine the value of the Content Master Key (CMK). The algorithm specified by the "alg" value MUST be supported by the implementation and there MUST be a key for use with that algorithm associated with the intended recipient or the JWE MUST be rejected. "alg" values SHOULD either be registered in the IANA JSON Web Signature and Encryption Algorithms registry [JWA] or be a URI that contains a Collision Resistant Namespace. The "alg" value is a case sensitive string containing a StringOrURI value. This header parameter is REQUIRED.

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

4.1.2. "enc" (Encryption Method) Header Parameter

The "enc" (encryption method) header parameter identifies the symmetric encryption algorithm used to encrypt the Plaintext to produce the Ciphertext. The algorithm specified by the "enc" value MUST be supported by the implementation or the JWE MUST be rejected. "enc" values SHOULD either be registered in the IANA JSON Web Signature and Encryption Algorithms registry [JWA] or be a URI that contains a Collision Resistant Namespace. The "enc" value is a case sensitive string containing a StringOrURI value. This header parameter is REQUIRED.

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

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

4.1.3. "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] value. This header parameter is OPTIONAL, although its use is REQUIRED with some "alg" algorithms.

4.1.4. "zip" (Compression Algorithm) Header Parameter

The "zip" (compression algorithm) applied to the Plaintext before encryption, if any. If present, the value of the "zip" header parameter MUST be the case sensitive string "DEF". Compression is performed with the DEFLATE [RFC1951] algorithm. If no "zip" parameter is present, no compression is applied to the Plaintext before encryption. This header parameter is OPTIONAL.

4.1.5. "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 encrypt the JWE; this can be used to determine the private key needed to decrypt the JWE. 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 certificate MUST use TLS [RFC2818] [RFC5246]; the identity of the server MUST be validated, as per Section 3.1 of HTTP Over TLS [RFC2818]. This header parameter is OPTIONAL.

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

The "jwk" (JSON Web Key) header parameter is a public key that corresponds to the key used to encrypt the JWE; this can be used to determine the private key needed to decrypt the JWE. This key is represented as a JSON Web Key [JWK]. This header parameter is OPTIONAL.

4.1.7. "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 encrypt the JWE; this can be used to determine the private key needed to decrypt the JWE. The identified resource MUST provide a representation of the certificate or certificate chain that conforms to RFC 5280 [RFC5280] in PEM encoded form [RFC1421]. The certificate containing the public

key of the entity that encrypted the JWE MUST be the first certificate. This MAY be followed by additional certificates, with each subsequent certificate being the one used to certify the previous one. The protocol used to acquire the resource MUST provide integrity protection; an HTTP GET request to retrieve the certificate MUST use TLS [RFC2818] [RFC5246]; the identity of the server MUST be validated, as per Section 3.1 of HTTP Over TLS [RFC2818]. This header parameter is OPTIONAL.

4.1.8. "x5t" (X.509 Certificate Thumbprint) Header Parameter

The "x5t" (X.509 Certificate Thumbprint) header parameter provides 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 encrypt the JWE; this can be used to determine the private key needed to decrypt the JWE. This header parameter is OPTIONAL.

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

4.1.9. "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 encrypt the JWE; this can be used to determine the private key needed to decrypt the JWE. The certificate or certificate chain is represented as an array of certificate value strings. Each string is a base64 encoded ([RFC4648] Section 4 -- not base64url encoded) DER [ITU.X690.1994] PKIX certificate value. The certificate containing the public key of the entity that encrypted the JWE MUST be the first certificate. This MAY be followed by additional certificates, with each subsequent certificate being the one used to certify the previous one. The recipient MUST verify the certificate chain according to [RFC5280] and reject the JWE if any validation failure occurs. This header parameter is OPTIONAL.

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

4.1.10. "kid" (Key ID) Header Parameter

The "kid" (key ID) header parameter is a hint indicating which key was used to encrypt the JWE; 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 recipients. Should the recipient be unable to locate a key corresponding to the "kid" value, they SHOULD treat that condition as an error. The interpretation of the "kid" value is unspecified. Its value MUST be a string. This header parameter is OPTIONAL.

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

4.1.11. "typ" (Type) Header Parameter

The "typ" (type) header parameter is used to declare the type of this object. The type value "JWE" MAY be used to indicate that this object is a JWE. The "typ" value is a case sensitive string. This header parameter is OPTIONAL.

MIME Media Type [RFC2046] values MAY be used as "typ" values.

"typ" values SHOULD either be registered in the IANA JSON Web Signature and Encryption Type Values registry [JWS] or be a URI that contains a Collision Resistant Namespace.

4.1.12. "cty" (Content Type) Header Parameter

The "cty" (content type) header parameter is used to declare the type of the encrypted content (the Plaintext). The "cty" value is a case sensitive string. This header parameter is OPTIONAL.

The values used for the "cty" header parameter come from the same value space as the "typ" header parameter, with the same rules applying.

4.1.13. "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. This header parameter is OPTIONAL.

4.1.14. "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. This header parameter is OPTIONAL.

4.1.15. "epu" (Encryption PartyUInfo) Header Parameter

The "epu" (encryption PartyUInfo) value for plaintext encryption algorithms using it (such as "A128CBC+HS256"), represented as a

base64url encoded string. This header parameter is OPTIONAL.

4.1.16. "epv" (Encryption PartyVInfo) Header Parameter

The "epv" (encryption PartyVInfo) value for plaintext encryption algorithms using it (such as "A128CBC+HS256"), represented as a base64url encoded string. This header parameter is OPTIONAL.

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 [JWS] or be a URI that contains a Collision Resistant Namespace. 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 any header parameter name that is not a Reserved Name Section 4.1 or a Public Name Section 4.2. Unlike Public Names, these private names are subject to collision and should be used with caution.

5. 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. When key wrapping, key encryption, or key agreement with key wrapping are employed, generate a random Content Master Key (CMK). See RFC 4086 [RFC4086] for considerations on generating random values. The CMK MUST have a length equal to that required for the block encryption algorithm.
2. When key agreement is employed, use the key agreement algorithm to compute the value of the agreed upon key. When key agreement without key wrapping is employed, let the Content Master Key (CMK) be the agreed upon key. When key agreement with key wrapping is employed, the agreed upon key will be used to wrap the CMK.

3. When key wrapping, key encryption, or key agreement with key wrapping are employed, encrypt the CMK for the recipient (see Section 7) and let the result be the JWE Encrypted Key. Otherwise, when direct encryption with a shared or agreed upon symmetric key is employed, let the JWE Encrypted Key be the empty byte array.
4. When direct encryption with a shared symmetric key is employed, let the Content Master Key (CMK) be the shared key.
5. Base64url encode the JWE Encrypted Key to create the Encoded JWE Encrypted Key.
6. Generate a random JWE Initialization Vector of the correct size for the block encryption algorithm (if required for the algorithm); otherwise, let the JWE Initialization Vector be the empty byte string.
7. Base64url encode the JWE Initialization Vector to create the Encoded JWE Initialization Vector.
8. Compress the Plaintext if a "zip" parameter was included.
9. Serialize the (compressed) Plaintext into a byte sequence M.
10. Create a JWE Header containing the encryption parameters used. Note that white space is explicitly allowed in the representation and no canonicalization need be performed before encoding.
11. Base64url encode the bytes of the UTF-8 representation of the JWE Header to create the Encoded JWE Header.
12. Encrypt M using the CMK, the JWE Initialization Vector, and the other parameters required for the specified block encryption algorithm to create the JWE Ciphertext value and the JWE Integrity Value.
13. Base64url encode the JWE Ciphertext to create the Encoded JWE Ciphertext.
14. Base64url encode the JWE Integrity Value to create the Encoded JWE Integrity Value.
15. The five encoded parts, taken together, are the result.
16. The Compact Serialization of this result is the concatenation of the Encoded JWE Header, the Encoded JWE Encrypted Key, the

Encoded JWE Initialization Vector, the Encoded JWE Ciphertext, and the Encoded JWE Integrity Value in that order, with the five strings being separated by four period ('.') characters.

6. 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 JWE MUST be rejected.

1. Determine the Encoded JWE Header, the Encoded JWE Encrypted Key, the Encoded JWE Initialization Vector, the Encoded JWE Ciphertext, and the Encoded JWE Integrity Value values contained in the JWE. When using the Compact Serialization, these five values are represented in that order, separated by four period ('.') characters.
2. The Encoded JWE Header, the Encoded JWE Encrypted Key, the Encoded JWE Initialization Vector, the Encoded JWE Ciphertext, and the Encoded JWE Integrity Value MUST be successfully base64url decoded following the restriction that no padding characters have been used.
3. The resulting JWE Header MUST be completely valid JSON syntax conforming to RFC 4627 [RFC4627].
4. The resulting JWE Header MUST be validated to only include parameters and values whose syntax and semantics are both understood and supported.
5. Verify that the JWE uses a key known to the recipient.
6. When key agreement is employed, use the key agreement algorithm to compute the value of the agreed upon key. When key agreement without key wrapping is employed, let the Content Master Key (CMK) 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.
7. When key wrapping, key encryption, or key agreement with key wrapping are employed, decrypt the JWE Encrypted Key to produce the Content Master Key (CMK). The CMK MUST have a length equal to that required for the block encryption algorithm.
8. When direct encryption with a shared symmetric key is employed, let the Content Master Key (CMK) be the shared key.

9. Decrypt the JWE Ciphertext using the CMK, the JWE Initialization Vector, and the other parameters required for the specified block encryption algorithm, returning the decrypted plaintext and verifying the JWE Integrity Value in the manner specified for the algorithm.
10. Uncompress the decrypted plaintext if a "zip" parameter was included.
11. Output the resulting Plaintext.

7. CMK Encryption

JWE supports three forms of Content Master Key (CMK) encryption:

- o Asymmetric encryption under the recipient's public key.
- o Symmetric encryption under a key shared between the sender and receiver.
- o Symmetric encryption under a key agreed upon between the sender and receiver.

See the algorithms registered for "enc" usage in the IANA JSON Web Signature and Encryption Algorithms registry [JWA] and Section 4.1 of the JSON Web Algorithms (JWA) [JWA] specification for lists of encryption algorithms that can be used for CMK encryption.

8. Encrypting JWEs with Cryptographic Algorithms

JWE uses cryptographic algorithms to encrypt the Plaintext and the Content Encryption Key (CMK) and to provide integrity protection for the JWE Header, JWE Encrypted Key, and JWE Ciphertext. The JSON Web Algorithms (JWA) [JWA] specification specifies a set of cryptographic algorithms and identifiers to be used with this specification and defines registries for additional such algorithms. Specifically, Section 4.1 specifies a set of "alg" (algorithm) header parameter values and Section 4.2 specifies a set of "enc" (encryption method) header parameter values intended for use this specification. It also describes the semantics and operations that are specific to these algorithms and algorithm families.

Public keys employed for encryption can be identified using the Header Parameter methods described in Section 4.1 or can be distributed using methods that are outside the scope of this specification.

9. IANA Considerations

9.1. Registration of JWE Header Parameter Names

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

9.1.1. Registry Contents

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

- o Header Parameter Name: "enc"
- o Change Controller: IETF
- o Specification Document(s): Section 4.1.2 of [[this document]]

- o Header Parameter Name: "epk"
- o Change Controller: IETF
- o Specification Document(s): Section 4.1.3 of [[this document]]

- o Header Parameter Name: "zip"
- o Change Controller: IETF
- o Specification Document(s): Section 4.1.4 of [[this document]]

- o Header Parameter Name: "jku"
- o Change Controller: IETF
- o Specification Document(s): Section 4.1.5 of [[this document]]

- o Header Parameter Name: "jwk"
- o Change Controller: IETF
- o Specification document(s): Section 4.1.6 of [[this document]]

- o Header Parameter Name: "x5u"
- o Change Controller: IETF
- o Specification Document(s): Section 4.1.7 of [[this document]]

- o Header Parameter Name: "x5t"
- o Change Controller: IETF
- o Specification Document(s): Section 4.1.8 of [[this document]]

- o Header Parameter Name: "x5c"
- o Change Controller: IETF
- o Specification Document(s): Section 4.1.9 of [[this document]]

- o Header Parameter Name: "kid"
- o Change Controller: IETF
- o Specification Document(s): Section 4.1.10 of [[this document]]
- o Header Parameter Name: "typ"
- o Change Controller: IETF
- o Specification Document(s): Section 4.1.11 of [[this document]]
- o Header Parameter Name: "cty"
- o Change Controller: IETF
- o Specification Document(s): Section 4.1.12 of [[this document]]
- o Header Parameter Name: "apu"
- o Change Controller: IETF
- o Specification Document(s): Section 4.1.13 of [[this document]]
- o Header Parameter Name: "apv"
- o Change Controller: IETF
- o Specification Document(s): Section 4.1.14 of [[this document]]
- o Header Parameter Name: "epu"
- o Change Controller: IETF
- o Specification Document(s): Section 4.1.15 of [[this document]]
- o Header Parameter Name: "epv"
- o Change Controller: IETF
- o Specification Document(s): Section 4.1.16 of [[this document]]

9.2. JSON Web Signature and Encryption Type Values Registration

9.2.1. Registry Contents

This specification registers the "JWE" type value in the IANA JSON Web Signature and Encryption Type Values registry [JWS]:

- o "typ" Header Parameter Value: "JWE"
- o Abbreviation for MIME Type: application/jwe
- o Change Controller: IETF
- o Specification Document(s): Section 4.1.11 of [[this document]]

9.3. Media Type Registration

9.3.1. Registry Contents

This specification registers the "application/jwe" Media Type [RFC2046] in the MIME Media Type registry [RFC4288] to indicate that the content is a JWE using the Compact Serialization.

- o Type Name: application
- o Subtype Name: jwe
- o Required Parameters: n/a
- o Optional Parameters: n/a
- o Encoding considerations: JWE values are encoded as a series of base64url encoded values (some of which may be the empty string) separated by period ('.') characters
- o Security Considerations: See the Security Considerations section of this document
- o Interoperability Considerations: n/a
- o Published Specification: [[this document]]
- o Applications that use this media type: OpenID Connect and other applications using encrypted JWTs
- o Additional Information: Magic number(s): n/a, File extension(s): n/a, Macintosh file type code(s): n/a
- o Person & email address to contact for further information: Michael B. Jones, mbj@microsoft.com
- o Intended Usage: COMMON
- o Restrictions on Usage: none
- o Author: Michael B. Jones, mbj@microsoft.com
- o Change Controller: IETF

10. Security Considerations

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

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

11. References

11.1. Normative References

[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)", October 2012.
- [JWK] Jones, M., "JSON Web Key (JWK)", October 2012.
- [JWS] Jones, M., Bradley, J., and N. Sakimura, "JSON Web Signature (JWS)", October 2012.
- [RFC1421] Linn, J., "Privacy Enhancement for Internet Electronic Mail: Part I: Message Encryption and Authentication Procedures", RFC 1421, February 1993.
- [RFC1951] Deutsch, P., "DEFLATE Compressed Data Format Specification version 1.3", RFC 1951, May 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.
- [RFC4086] Eastlake, D., Schiller, J., and S. Crocker, "Randomness Requirements for Security", BCP 106, RFC 4086, June 2005.
- [RFC4288] Freed, N. and J. Klensin, "Media Type Specifications and Registration Procedures", BCP 13, RFC 4288, December 2005.
- [RFC4627] Crockford, D., "The application/json Media Type for JavaScript Object Notation (JSON)", RFC 4627, July 2006.
- [RFC4648] Josefsson, S., "The Base16, Base32, and Base64 Data Encodings", RFC 4648, October 2006.
- [RFC5116] McGrew, D., "An Interface and Algorithms for Authenticated Encryption", RFC 5116, January 2008.
- [RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security

(TLS) Protocol Version 1.2", RFC 5246, August 2008.

[RFC5280] Cooper, D., Santesson, S., Farrell, S., Boeyen, S., Housley, R., and W. Polk, "Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile", RFC 5280, May 2008.

[W3C.CR-xmlenc-core1-20120313]
Eastlake, D., Reagle, J., Hirsch, F., and T. Roessler,
"XML Encryption Syntax and Processing Version 1.1", World
Wide Web Consortium CR CR-xmlenc-core1-20120313,
March 2012,
<<http://www.w3.org/TR/2012/CR-xmlenc-core1-20120313>>.

11.2. Informative References

- [I-D.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.
- [JWE-JS] Jones, M., "JSON Web Encryption JSON Serialization (JWE-JS)", October 2012.
- [RFC4122] Leach, P., Mealling, M., and R. Salz, "A Universally Unique Identifier (UUID) URN Namespace", RFC 4122, July 2005.
- [RFC5652] Housley, R., "Cryptographic Message Syntax (CMS)", STD 70, RFC 5652, September 2009.

Appendix A. JWE Examples

This section provides examples of JWE computations.

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

This example encrypts the plaintext "Live long and prosper." to the recipient using RSAES OAEP and AES GCM. The AES GCM algorithm has an integrated integrity check. The representation of this plaintext is:

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

A.1.1. JWE Header

The following example JWE Header declares that:

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

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

A.1.2. Encoded JWE Header

Base64url encoding the bytes of the UTF-8 representation of the JWE Header yields this Encoded JWE Header value:

```
eyJhbGciOiJSU0EtT0FFUCIsImVuYyI6IkeYNTZHQ00ifQ
```

A.1.3. Content Master Key (CMK)

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

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

A.1.4. Key Encryption

Encrypt the CMK with the recipient's public key using the RSAES OAEP algorithm to produce the JWE Encrypted Key. In this example, the RSA key parameters are:

Parameter Name	Value
Modulus	[161, 168, 84, 34, 133, 176, 208, 173, 46, 176, 163, 110, 57, 30, 135, 227, 9, 31, 226, 128, 84, 92, 116, 241, 70, 248, 27, 227, 193, 62, 5, 91, 241, 145, 224, 205, 141, 176, 184, 133, 239, 43, 81, 103, 9, 161, 153, 157, 179, 104, 123, 51, 189, 34, 152, 69, 97, 69, 78, 93, 140, 131, 87, 182, 169, 101, 92, 142, 3, 22, 167, 8, 212, 56, 35, 79, 210, 222, 192, 208, 252, 49, 109, 138, 173, 253, 210, 166, 201, 63, 102, 74, 5, 158, 41, 90, 144, 108, 160, 79, 10, 89, 222, 231, 172, 31, 227, 197, 0, 19, 72, 81, 138, 78, 136, 221, 121, 118, 196, 17, 146, 10, 244, 188, 72, 113, 55, 221, 162, 217, 171, 27, 57, 233, 210, 101, 236, 154, 199, 56, 138, 239, 101, 48, 198, 186, 202, 160, 76, 111, 234, 71, 57, 183, 5, 211, 171, 136, 126, 64, 40, 75, 58, 89, 244, 254, 107, 84, 103, 7, 236, 69, 163, 18, 180, 251, 58, 153, 46, 151, 174, 12, 103, 197, 181, 161, 162, 55, 250, 235, 123, 110, 17, 11, 158, 24, 47, 133, 8, 199, 235, 107, 126, 130, 246, 73, 195, 20, 108, 202, 176, 214, 187, 45, 146, 182, 118, 54, 32, 200, 61, 201, 71, 243, 1, 255, 131, 84, 37, 111, 211, 168, 228, 45, 192, 118, 27, 197, 235, 232, 36, 10, 230, 248, 190, 82, 182, 140, 35, 204, 108, 190, 253, 186, 186, 27]
Exponent	[1, 0, 1]

Private Exponent	[144, 183, 109, 34, 62, 134, 108, 57, 44, 252, 10, 66, 73, 54, 16, 181, 233, 92, 54, 219, 101, 42, 35, 178, 63, 51, 43, 92, 119, 136, 251, 41, 53, 23, 191, 164, 164, 60, 88, 227, 229, 152, 228, 213, 149, 228, 169, 237, 104, 71, 151, 75, 88, 252, 216, 77, 251, 231, 28, 97, 88, 193, 215, 202, 248, 216, 121, 195, 211, 245, 250, 112, 71, 243, 61, 129, 95, 39, 244, 122, 225, 217, 169, 211, 165, 48, 253, 220, 59, 122, 219, 42, 86, 223, 32, 236, 39, 48, 103, 78, 122, 216, 187, 88, 176, 89, 24, 1, 42, 177, 24, 99, 142, 170, 1, 146, 43, 3, 108, 64, 194, 121, 182, 95, 187, 134, 71, 88, 96, 134, 74, 131, 167, 69, 106, 143, 121, 27, 72, 44, 245, 95, 39, 194, 179, 175, 203, 122, 16, 112, 183, 17, 200, 202, 31, 17, 138, 156, 184, 210, 157, 184, 154, 131, 128, 110, 12, 85, 195, 122, 241, 79, 251, 229, 183, 117, 21, 123, 133, 142, 220, 153, 9, 59, 57, 105, 81, 255, 138, 77, 82, 54, 62, 216, 38, 249, 208, 17, 197, 49, 45, 19, 232, 157, 251, 131, 137, 175, 72, 126, 43, 229, 69, 179, 117, 82, 157, 213, 83, 35, 57, 210, 197, 252, 171, 143, 194, 11, 47, 163, 6, 253, 75, 252, 96, 11, 187, 84, 130, 210, 7, 121, 78, 91, 79, 57, 251, 138, 132, 220, 60, 224, 173, 56, 224, 201]
------------------	---

The resulting JWE Encrypted Key value is:

[51, 101, 241, 165, 179, 145, 41, 236, 202, 75, 60, 208, 47, 255, 121, 248, 104, 226, 185, 212, 65, 78, 169, 255, 162, 100, 188, 207, 220, 96, 161, 22, 251, 47, 66, 112, 229, 75, 4, 111, 25, 173, 200, 121, 246, 79, 189, 102, 173, 146, 228, 142, 14, 12, 167, 200, 27, 133, 138, 37, 180, 249, 4, 56, 123, 192, 162, 156, 246, 231, 235, 217, 240, 45, 158, 213, 195, 154, 2, 142, 86, 61, 198, 210, 34, 225, 92, 7, 128, 227, 4, 227, 55, 183, 69, 0, 59, 162, 71, 145, 98, 238, 0, 70, 40, 123, 159, 37, 115, 18, 16, 157, 236, 138, 117, 166, 18, 45, 181, 125, 112, 170, 168, 82, 129, 80, 166, 242, 150, 97, 17, 217, 109, 251, 51, 35, 39, 236, 107, 95, 43, 154, 4, 227, 206, 187, 75, 13, 51, 231, 115, 79, 67, 72, 145, 54, 225, 164, 60, 195, 120, 188, 69, 113, 3, 182, 21, 189, 79, 82, 122, 46, 196, 199, 254, 252, 7, 119, 5, 32, 144, 143, 173, 11, 99, 205, 120, 106, 231, 51, 231, 77, 73, 252, 197, 221, 142, 254, 151, 7, 6, 203, 65, 108, 117, 121, 15, 95, 43, 111, 13, 94, 242, 226, 150, 94, 121, 72, 144, 251, 69, 93, 137, 178, 13, 216, 8, 227, 125, 110, 180, 157, 250, 207, 184, 232, 222, 164, 193, 70, 232, 16, 65, 109, 29, 251, 164, 119, 50, 205, 236, 109, 245, 234, 78, 1]

A.1.1.5. Encoded JWE Encrypted Key

Base64url encode the JWE Encrypted Key to produce the Encoded JWE Encrypted Key. This result (with line breaks for display purposes only) is:

```
M2XxpbORKezKSzzQL_95-GjiudRBTqn_omS8z9xgoRb7L0Jw5UsEbxmtyHn2T71m
rZLkjpg4Mp8gbhYoltPkEOHvAopz25-vZ8C2e1cOaAo5WPcbSiUfCB4DjBOM3t0UA
O6JHkWLuAEYoe58lcxIQneyKdaYSLbV9cKqoUoFQpvKWYRHZbfszIyfsa18rmgTj
zrtLDTPnc09DSJE24aQ8w3i8RXEDthW9TlJ6LsTH_vwHdwUgkI-tC2PNeGrnM-dN
SfzF3Y7-lwcGy0FsdXkPXyTvDV7y4pZeeUiQ-0VdibIN2AjjfW60nfrPuOjepMFG
6BBBbR37pHcyzext9epOAQ
```

A.1.1.6. 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]
```

Base64url encoding this value yields the Encoded JWE Initialization Vector value:

```
48Vl_ALb6US04U3b
```

A.1.1.7. "Additional Authenticated Data" Parameter

Concatenate the Encoded JWE Header value, a period character ('.'), the Encoded JWE Encrypted Key, a second period character ('.'), and the Encoded JWE Initialization Vector to create the "additional authenticated data" parameter for the AES GCM algorithm. This result (with line breaks for display purposes only) is:

```
eyJhbGciOiJSU0EtT0FFUCIsImVuYyI6IkJNTZHQ00ifQ.
M2XxpbORKezKSzzQL_95-GjiudRBTqn_omS8z9xgoRb7L0Jw5UsEbxmtyHn2T71m
rZLkjpg4Mp8gbhYoltPkEOHvAopz25-vZ8C2e1cOaAo5WPcbSiUfCB4DjBOM3t0UA
O6JHkWLuAEYoe58lcxIQneyKdaYSLbV9cKqoUoFQpvKWYRHZbfszIyfsa18rmgTj
zrtLDTPnc09DSJE24aQ8w3i8RXEDthW9TlJ6LsTH_vwHdwUgkI-tC2PNeGrnM-dN
SfzF3Y7-lwcGy0FsdXkPXyTvDV7y4pZeeUiQ-0VdibIN2AjjfW60nfrPuOjepMFG
6BBBbR37pHcyzext9epOAQ.
48Vl_ALb6US04U3b
```

The representation of 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, 46,
77, 50, 88, 120, 112, 98, 79, 82, 75, 101, 122, 75, 83, 122, 122, 81,
```

76, 95, 57, 53, 45, 71, 106, 105, 117, 100, 82, 66, 84, 113, 110, 95, 111, 109, 83, 56, 122, 57, 120, 103, 111, 82, 98, 55, 76, 48, 74, 119, 53, 85, 115, 69, 98, 120, 109, 116, 121, 72, 110, 50, 84, 55, 49, 109, 114, 90, 76, 107, 106, 103, 52, 77, 112, 56, 103, 98, 104, 89, 111, 108, 116, 80, 107, 69, 79, 72, 118, 65, 111, 112, 122, 50, 53, 45, 118, 90, 56, 67, 50, 101, 49, 99, 79, 97, 65, 111, 53, 87, 80, 99, 98, 83, 73, 117, 70, 99, 66, 52, 68, 106, 66, 79, 77, 51, 116, 48, 85, 65, 79, 54, 74, 72, 107, 87, 76, 117, 65, 69, 89, 111, 101, 53, 56, 108, 99, 120, 73, 81, 110, 101, 121, 75, 100, 97, 89, 83, 76, 98, 86, 57, 99, 75, 113, 111, 85, 111, 70, 81, 112, 118, 75, 87, 89, 82, 72, 90, 98, 102, 115, 122, 73, 121, 102, 115, 97, 49, 56, 114, 109, 103, 84, 106, 122, 114, 116, 76, 68, 84, 80, 110, 99, 48, 57, 68, 83, 74, 69, 50, 52, 97, 81, 56, 119, 51, 105, 56, 82, 88, 69, 68, 116, 104, 87, 57, 84, 49, 74, 54, 76, 115, 84, 72, 95, 118, 119, 72, 100, 119, 85, 103, 107, 73, 45, 116, 67, 50, 80, 78, 101, 71, 114, 110, 77, 45, 100, 78, 83, 102, 122, 70, 51, 89, 55, 45, 108, 119, 99, 71, 121, 48, 70, 115, 100, 88, 107, 80, 88, 121, 116, 118, 68, 86, 55, 121, 52, 112, 90, 101, 101, 85, 105, 81, 45, 48, 86, 100, 105, 98, 73, 78, 50, 65, 106, 106, 102, 87, 54, 48, 110, 102, 114, 80, 117, 79, 106, 101, 112, 77, 70, 71, 54, 66, 66, 66, 98, 82, 51, 55, 112, 72, 99, 121, 122, 101, 120, 116, 57, 101, 112, 79, 65, 81, 46, 52, 56, 86, 49, 95, 65, 76, 98, 54, 85, 83, 48, 52, 85, 51, 98]

A.1.1.8. Plaintext Encryption

Encrypt the Plaintext with AES GCM using the CMK 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:

[253, 237, 181, 180, 97, 161, 105, 207, 233, 120, 65, 100, 45, 122, 246, 116, 195, 212, 102, 37, 36, 175]

The resulting "authentication tag" value is:

[237, 94, 89, 14, 74, 52, 191, 249, 159, 216, 240, 28, 224, 147, 34, 82]

A.1.1.9. Encoded JWE Ciphertext

Base64url encode the resulting Ciphertext to create the Encoded JWE Ciphertext. This result is:

e21tGGhac_peEFkLXr2dMPUziUkrw

A.1.10. Encoded JWE Integrity Value

Base64url encode the resulting "authentication tag" to create the Encoded JWE Integrity Value. This result is:

```
7V5ZDko0v_mf2Pac4JMiUg
```

A.1.11. Complete Representation

Assemble the final representation: The Compact Serialization of this result is the concatenation of the Encoded JWE Header, the Encoded JWE Encrypted Key, the Encoded JWE Initialization Vector, the Encoded JWE Ciphertext, and the Encoded JWE Integrity Value in that order, with the five strings being separated by four period ('.') characters.

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

```
eyJhbGciOiJSU0EtT0FFUCIsImVuYyI6IkJNTZHQ00ifQ.  
M2XxpBORKezKSzzQL_95-GjiudRBTqn_omS8z9xgoRb7L0Jw5UsEbxmtyHn2T7lm  
rZLkjg4Mp8gbhYoltPkEOHvAopz25-vZ8C2elcOaAo5WPcbSIuFcB4DjBOM3t0UA  
O6JHkWLuAEYoe58lclxIQneyKdaYSLbV9cKqoUoFQpvKWYRHZbfszIyfsal8rmgTj  
zrtLDTPnc09DSJE24aQ8w3i8RXEDthW9TlJ6LsTH_vwHdwUgkI-tC2PNeGrnM-dN  
SfzF3Y7-1wcGy0FsdXkPXytdV7y4pZeeUiQ-0VdibIN2AjjfW60nfrPuOjepMFG  
6BBBbR37pHcyzext9epOAQ.  
48V1_ALb6US04U3b.  
_e2ltGGhac_peEFkLXr2dMPUziUkrw.  
7V5ZDko0v_mf2Pac4JMiUg
```

A.1.12. Validation

This example illustrates the process of creating a JWE with an AEAD algorithm. 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 CBC

This example encrypts the plaintext "No matter where you go, there you are." to the recipient using RSAES-PKCS1-V1_5 and AES CBC. AES CBC does not have an integrated integrity check, so a separate integrity check calculation is performed using HMAC SHA-256, with separate encryption and integrity keys being derived from a master key using the Concat KDF with the SHA-256 digest function. The

representation of this plaintext is:

```
[78, 111, 32, 109, 97, 116, 116, 101, 114, 32, 119, 104, 101, 114,
101, 32, 121, 111, 117, 32, 103, 111, 44, 32, 116, 104, 101, 114,
101, 32, 121, 111, 117, 32, 97, 114, 101, 46]
```

A.2.1. JWE Header

The following example JWE Header (with line breaks for display purposes only) declares that:

- o the Content Master Key is encrypted to the recipient using the RSAES-PKCS1-V1_5 algorithm to produce the JWE Encrypted Key and
- o the Plaintext is encrypted using the AES CBC algorithm with a 128 bit key to produce the Ciphertext, with the integrity of the Ciphertext and the parameters used to create it being secured with the HMAC SHA-256 algorithm.

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

A.2.2. Encoded JWE Header

Base64url encoding the bytes of the UTF-8 representation of the JWE Header yields this Encoded JWE Header value:

```
eyJhbGciOiJSU0ExXzUiLCJlbmMiOiJBMTI4Q0JDK0hTMjU2In0
```

A.2.3. Content Master Key (CMK)

Generate a 256 bit random Content Master Key (CMK). 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.2.4. Key Encryption

Encrypt the CMK with the recipient's public key using the RSAES-PKCS1-V1_5 algorithm to produce the JWE Encrypted Key. In this example, the RSA key parameters are:

Parameter Name	Value
Modulus	[177, 119, 33, 13, 164, 30, 108, 121, 207, 136, 107, 242, 12, 224, 19, 226, 198, 134, 17, 71, 173, 75, 42, 61, 48, 162, 206, 161, 97, 108, 185, 234, 226, 219, 118, 206, 118, 5, 169, 224, 60, 181, 90, 85, 51, 123, 6, 224, 4, 122, 29, 230, 151, 12, 244, 127, 121, 25, 4, 85, 220, 144, 215, 110, 130, 17, 68, 228, 129, 138, 7, 130, 231, 40, 212, 214, 17, 179, 28, 124, 151, 178, 207, 20, 14, 154, 222, 113, 176, 24, 198, 73, 211, 113, 9, 33, 178, 80, 13, 25, 21, 25, 153, 212, 206, 67, 154, 147, 70, 194, 192, 183, 160, 83, 98, 236, 175, 85, 23, 97, 75, 199, 177, 73, 145, 50, 253, 206, 32, 179, 254, 236, 190, 82, 73, 67, 129, 253, 252, 220, 108, 136, 138, 11, 192, 1, 36, 239, 228, 55, 81, 113, 17, 25, 140, 63, 239, 146, 3, 172, 96, 60, 227, 233, 64, 255, 224, 173, 225, 228, 229, 92, 112, 72, 99, 97, 26, 87, 187, 123, 46, 50, 90, 202, 117, 73, 10, 153, 47, 224, 178, 163, 77, 48, 46, 154, 33, 148, 34, 228, 33, 172, 216, 89, 46, 225, 127, 68, 146, 234, 30, 147, 54, 146, 5, 133, 45, 78, 254, 85, 55, 75, 213, 86, 194, 218, 215, 163, 189, 194, 54, 6, 83, 36, 18, 153, 53, 7, 48, 89, 35, 66, 144, 7, 65, 154, 13, 97, 75, 55, 230, 132, 3, 13, 239, 71]
Exponent	[1, 0, 1]

Private Exponent	[84, 80, 150, 58, 165, 235, 242, 123, 217, 55, 38, 154, 36, 181, 221, 156, 211, 215, 100, 164, 90, 88, 40, 228, 83, 148, 54, 122, 4, 16, 165, 48, 76, 194, 26, 107, 51, 53, 179, 165, 31, 18, 198, 173, 78, 61, 56, 97, 252, 158, 140, 80, 63, 25, 223, 156, 36, 203, 214, 252, 120, 67, 180, 167, 3, 82, 243, 25, 97, 214, 83, 133, 69, 16, 104, 54, 160, 200, 41, 83, 164, 187, 70, 153, 111, 234, 242, 158, 175, 28, 198, 48, 211, 45, 148, 58, 23, 62, 227, 74, 52, 117, 42, 90, 41, 249, 130, 154, 80, 119, 61, 26, 193, 40, 125, 10, 152, 174, 227, 225, 205, 32, 62, 66, 6, 163, 100, 99, 219, 19, 253, 25, 105, 80, 201, 29, 252, 157, 237, 69, 1, 80, 171, 167, 20, 196, 156, 109, 249, 88, 0, 3, 152, 38, 165, 72, 87, 6, 152, 71, 156, 214, 16, 71, 30, 82, 51, 103, 76, 218, 63, 9, 84, 163, 249, 91, 215, 44, 238, 85, 101, 240, 148, 1, 82, 224, 91, 135, 105, 127, 84, 171, 181, 152, 210, 183, 126, 24, 46, 196, 90, 173, 38, 245, 219, 186, 222, 27, 240, 212, 194, 15, 66, 135, 226, 178, 190, 52, 245, 74, 65, 224, 81, 100, 85, 25, 204, 165, 203, 187, 175, 84, 100, 82, 15, 11, 23, 202, 151, 107, 54, 41, 207, 3, 136, 229, 134, 131, 93, 139, 50, 182, 204, 93, 130, 89]
------------------	--

The resulting JWE Encrypted Key value is:

[59, 160, 42, 94, 168, 21, 148, 146, 120, 115, 137, 105, 230, 197, 217, 119, 182, 233, 24, 112, 48, 232, 4, 100, 29, 71, 151, 67, 16, 245, 112, 5, 184, 249, 125, 113, 210, 171, 99, 252, 3, 116, 154, 233, 42, 16, 233, 120, 99, 165, 14, 192, 149, 200, 37, 143, 126, 27, 81, 11, 43, 228, 80, 90, 178, 135, 253, 43, 81, 248, 3, 61, 31, 177, 169, 42, 100, 132, 142, 47, 16, 96, 231, 12, 58, 58, 254, 187, 209, 245, 46, 223, 233, 4, 72, 158, 27, 205, 54, 75, 21, 179, 210, 182, 197, 77, 116, 92, 143, 128, 86, 47, 23, 41, 10, 216, 212, 103, 125, 64, 63, 235, 28, 248, 191, 222, 9, 158, 84, 54, 93, 26, 73, 19, 106, 22, 201, 46, 62, 76, 209, 176, 86, 81, 12, 154, 146, 72, 151, 149, 59, 34, 123, 165, 144, 144, 243, 119, 229, 139, 95, 85, 130, 47, 247, 71, 181, 204, 82, 169, 110, 66, 235, 156, 237, 145, 206, 90, 44, 98, 227, 184, 215, 148, 20, 229, 89, 22, 121, 136, 92, 90, 75, 193, 186, 39, 204, 20, 173, 222, 1, 42, 8, 0, 195, 122, 151, 229, 122, 8, 125, 116, 21, 149, 6, 68, 209, 89, 164, 171, 252, 208, 72, 138, 57, 46, 113, 74, 202, 140, 40, 161, 23, 29, 176, 75, 141, 135, 208, 125, 50, 76, 173, 209, 148, 4, 148, 111, 163, 25, 219, 71, 90, 54, 82, 56]

A.2.5. Encoded JWE Encrypted Key

Base64url encode the JWE Encrypted Key to produce the Encoded JWE Encrypted Key. This result (with line breaks for display purposes only) is:

```
O6AqXqgVlJJ4c4lp5sXZd7bpGHaw6ARkHUeXQxDlcAW4-X1x0qtj_AN0mukqE0l4
Y6UOwJXlJY9-G1ELK-RQWrKH_StR-AM9H7GpKmSEji8QYOcmOjr-u9H1Lt_pBEie
G802SxWz0rbFTXRcj4BWLxcpCtjUZ3lAP-sc-L_eCZ5UNl0aSRNqFskuPkzRsFZR
DJqSSJeVOyJ7pZCQ83fli19Vgi_3R7XMUqluQuuc7ZHOWixi47jXlBTlWRZ5iFxa
S8G6J8wUrd4BKggAw3qX5XoIfXQVlQZE0Vmkq_zQSIo5LnFKyowooRcdsEuNh9B9
Mkyt0ZQE1G-jGdtHWjZSOA
```

A.2.6. Key Derivation

Use the Concat key derivation function to derive Content Encryption Key (CEK) and Content Integrity Key (CIK) values from the CMK. The details of this derivation are shown in Appendix A.4. The resulting CEK value is:

```
[37, 245, 125, 247, 113, 155, 238, 98, 228, 206, 62, 65, 81, 153, 79,
91]
```

The resulting CIK value is:

```
[203, 194, 197, 180, 120, 46, 123, 202, 78, 12, 33, 116, 214, 247,
128, 41, 175, 53, 181, 164, 224, 223, 56, 146, 179, 193, 18, 223,
146, 85, 244, 127]
```

A.2.7. 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]
```

Base64url encoding this value yields the Encoded JWE Initialization Vector value:

```
AxY8DctDaGlsbGljb3RoZQ
```

A.2.8. Plaintext Encryption

Encrypt the Plaintext with AES CBC using the CEK and the JWE Initialization Vector to produce the Ciphertext. The resulting Ciphertext is:

[213, 224, 86, 22, 7, 43, 207, 141, 48, 11, 207, 28, 130, 255, 43, 62, 11, 183, 17, 249, 130, 214, 158, 51, 79, 73, 8, 199, 23, 210, 23, 108, 195, 37, 196, 62, 136, 65, 228, 214, 52, 12, 247, 156, 64, 118, 190, 89]

A.2.9. Encoded JWE Ciphertext

Base64url encode the resulting Ciphertext to create the Encoded JWE Ciphertext. This result is:

```
1eBWFgcrz40wC88cgv8rPgu3EfmClp4zT0kIxxfSF2zDJcQ-iEHkljQM95xAdr5Z
```

A.2.10. Secured Input Value

Concatenate the Encoded JWE Header value, a period character ('.'), the Encoded JWE Encrypted Key, a second period character, the Encoded JWE Initialization Vector, a third period ('.') character, and the Encoded JWE Ciphertext to create the value to integrity protect. This result (with line breaks for display purposes only) is:

```
eyJhbGciOiJSU0ExXzUiLCJlbmMiOiJBMTI4Q0JDK0hTMjU2In0.
O6AqXqgVlJJ4c4lp5sXZd7bpGHAW6ARKHUEXQxDlcAW4-Xlx0qtj_AN0mukqEOl4
Y6UOWJXIJY9-G1ELK-RQWrKH_StR-AM9H7GpKmSEji8QYOcmOjr-u9H1Lt_pBEie
G802SxWz0rbFTXRcj4BWLxcpCtjUZ3lAP-sc-L_eCZ5UNl0aSRNqFskuPkzRsFZR
DJqSSJeVOyJ7pZCQ83fli19Vgi_3R7XMUqluQuuc7ZHOWixi47jXlBTlWRZ5iFxa
S8G6J8wUrd4BKggAw3qX5XoIfXQVlQZE0Vmkq_zQSIo5LnFKyowooRcdsEuNh9B9
Mkyt0ZQE1G-jGdtHWjZSOA.
AxY8DctDaGlsbGljb3RoZQ.
1eBWFgcrz40wC88cgv8rPgu3EfmClp4zT0kIxxfSF2zDJcQ-iEHkljQM95xAdr5Z
```

The representation of 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, 75, 48, 104, 84, 77, 106, 85, 50, 73, 110, 48, 46, 79, 54, 65, 113, 88, 113, 103, 86, 108, 74, 74, 52, 99, 52, 108, 112, 53, 115, 88, 90, 100, 55, 98, 112, 71, 72, 65, 119, 54, 65, 82, 107, 72, 85, 101, 88, 81, 120, 68, 49, 99, 65, 87, 52, 45, 88, 49, 120, 48, 113, 116, 106, 95, 65, 78, 48, 109, 117, 107, 113, 69, 79, 108, 52, 89, 54, 85, 79, 119, 74, 88, 73, 74, 89, 57, 45, 71, 49, 69, 76, 75, 45, 82, 81, 87, 114, 75, 72, 95, 83, 116, 82, 45, 65, 77, 57, 72, 55, 71, 112, 75, 109, 83, 69, 106, 105, 56, 81, 89, 79, 99, 77, 79, 106, 114, 45, 117, 57, 72, 49, 76, 116, 95, 112, 66, 69, 105, 101, 71, 56, 48, 50, 83, 120, 87, 122, 48, 114, 98, 70, 84, 88, 82, 99, 106, 52, 66, 87, 76, 120, 99, 112, 67, 116, 106, 85, 90, 51, 49, 65, 80, 45, 115, 99, 45, 76, 95, 101, 67, 90, 53, 85, 78, 108, 48, 97, 83, 82, 78, 113, 70, 115, 107, 117, 80, 107, 122, 82, 115, 70, 90, 82, 68, 74, 113, 83, 83, 74, 101, 86, 79, 121, 74,

55, 112, 90, 67, 81, 56, 51, 102, 108, 105, 49, 57, 86, 103, 105, 95,
51, 82, 55, 88, 77, 85, 113, 108, 117, 81, 117, 117, 99, 55, 90, 72,
79, 87, 105, 120, 105, 52, 55, 106, 88, 108, 66, 84, 108, 87, 82, 90,
53, 105, 70, 120, 97, 83, 56, 71, 54, 74, 56, 119, 85, 114, 100, 52,
66, 75, 103, 103, 65, 119, 51, 113, 88, 53, 88, 111, 73, 102, 88, 81,
86, 108, 81, 90, 69, 48, 86, 109, 107, 113, 95, 122, 81, 83, 73, 111,
53, 76, 110, 70, 75, 121, 111, 119, 111, 111, 82, 99, 100, 115, 69,
117, 78, 104, 57, 66, 57, 77, 107, 121, 116, 48, 90, 81, 69, 108, 71,
45, 106, 71, 100, 116, 72, 87, 106, 90, 83, 79, 65, 46, 65, 120, 89,
56, 68, 67, 116, 68, 97, 71, 108, 115, 98, 71, 108, 106, 98, 51, 82,
111, 90, 81, 46, 49, 101, 66, 87, 70, 103, 99, 114, 122, 52, 48, 119,
67, 56, 56, 99, 103, 118, 56, 114, 80, 103, 117, 51, 69, 102, 109,
67, 49, 112, 52, 122, 84, 48, 107, 73, 120, 120, 102, 83, 70, 50,
122, 68, 74, 99, 81, 45, 105, 69, 72, 107, 49, 106, 81, 77, 57, 53,
120, 65, 100, 114, 53, 90]

A.2.11. JWE Integrity Value

Compute the HMAC SHA-256 of this value using the CIK to create the JWE Integrity Value. This result is:

[68, 17, 161, 99, 49, 60, 253, 198, 75, 30, 50, 106, 168, 123, 139,
135, 54, 224, 90, 2, 255, 193, 93, 203, 13, 37, 43, 113, 185, 14,
136, 128]

A.2.12. Encoded JWE Integrity Value

Base64url encode the resulting JWE Integrity Value to create the Encoded JWE Integrity Value. This result is:

RBGhYzE8_cZLHjJqqHuLhzbGwGL_wV3LDSUrcbkOiIA

A.2.13. Complete Representation

Assemble the final representation: The Compact Serialization of this result is the concatenation of the Encoded JWE Header, the Encoded JWE Encrypted Key, the Encoded JWE Initialization Vector, the Encoded JWE Ciphertext, and the Encoded JWE Integrity Value in that order, with the five strings being separated by four period ('.') characters.

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

```

eyJhbGciOiJSU0ExXzUiLCJlbmMiOiJBMTI4Q0JDK0hTMjU2In0.
O6AqXqgVlJJ4c4lp5sXZd7bpGHAW6ARkHUeXQxD1cAW4-X1x0qtj_AN0mukqE0l4
Y6UowJXIjY9-G1ELK-RQWrKH_StR-AM9H7GpKmSEji8QY0cMOjr-u9H1Lt_pBEie
G802SxWz0rbFTXRcj4BWLxcpCtjUZ3lAP-sc-L_eCZ5UNl0aSRNqFskuPkzRsFZR
DJqSSJeVOyJ7pZCQ83fli19Vgi_3R7XMUqluQuuc7ZHOWixi47jXlBTlWRZ5iFxa
S8G6J8wUrd4BKggAw3qX5XoIfXQVlQZE0Vmkq_zQSIo5LnFKyowooRcdsEuNh9B9
MkYt0ZQE1G-jGdtHWjZSOA.
AxY8DcTdaGlSbGljb3RoZQ.
1eBWFgcrz40wC88cgv8rPgu3EfmClp4zT0kIxxfSF2zDJcQ-iEHkljQM95xAdr5Z.
RBGhYzE8_cZLHjJqqHuLhzbgWgL_wV3LDSUrcbkOiIA

```

A.2.14. Validation

This example illustrates the process of creating a JWE with a composite AEAD algorithm created from a non-AEAD algorithm by adding a separate integrity check calculation. 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 GCM

This example encrypts the plaintext "The true sign of intelligence is not knowledge but imagination." to the recipient using AES Key Wrap and AES GCM. The representation of this plaintext is:

```

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

```

A.3.1. JWE Header

The following example JWE Header declares that:

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

```

{"alg":"A128KW","enc":"A128GCM"}

```


A.3.2. Encoded JWE Header

Base64url encoding the bytes of the UTF-8 representation of the JWE Header yields this Encoded JWE Header value:

```
eyJhbGciOiJBMTI4S1ciLCJlbmMiOiJBMTI4R0NNIn0
```

A.3.3. Content Master Key (CMK)

Generate a 128 bit random Content Master Key (CMK). In this example, the value is:

```
[64, 154, 239, 170, 64, 40, 195, 99, 19, 84, 192, 142, 192, 238, 207, 217]
```

A.3.4. Key Encryption

Encrypt the CMK with the shared symmetric key using the AES Key Wrap algorithm to produce the JWE Encrypted Key. In this example, the shared symmetric key value is:

```
[25, 172, 32, 130, 225, 114, 26, 181, 138, 106, 254, 192, 95, 133, 74, 82]
```

The resulting JWE Encrypted Key value is:

```
[164, 255, 251, 1, 64, 200, 65, 200, 34, 197, 81, 143, 43, 211, 240, 38, 191, 161, 181, 117, 119, 68, 44, 80]
```

A.3.5. Encoded JWE Encrypted Key

Base64url encode the JWE Encrypted Key to produce the Encoded JWE Encrypted Key. This result is:

```
pP_7AUDIQcgixVGPK9PwJr-htXV3RCxQ
```

A.3.6. Initialization Vector

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

```
[253, 220, 80, 25, 166, 152, 178, 168, 97, 99, 67, 89]
```

Base64url encoding this value yields the Encoded JWE Initialization Vector value:

```
_dxQGaaYsqhhY0NZ
```

A.3.7. "Additional Authenticated Data" Parameter

Concatenate the Encoded JWE Header value, a period character ('.'), the Encoded JWE Encrypted Key, a second period character ('.'), and the Encoded JWE Initialization Vector to create the "additional authenticated data" parameter for the AES GCM algorithm. This result (with line breaks for display purposes only) is:

```
eyJhbGciOiJBMTI4S1ciLCJlbmMiOiJBMTI4R0NNIn0.  
pP_7AUDIQcgixVGPK9PwJr-htXV3RCxQ.  
_dxQGaaYsqhhY0NZ
```

The representation of 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, 82, 48, 78, 78, 73, 110, 48, 46, 112, 80, 95, 55, 65,  
85, 68, 73, 81, 99, 103, 105, 120, 86, 71, 80, 75, 57, 80, 119, 74,  
114, 45, 104, 116, 88, 86, 51, 82, 67, 120, 81, 46, 95, 100, 120, 81,  
71, 97, 97, 89, 115, 113, 104, 104, 89, 48, 78, 90]
```

A.3.8. Plaintext Encryption

Encrypt the Plaintext with AES GCM using the CMK 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:

```
[227, 12, 89, 132, 185, 16, 248, 93, 145, 87, 53, 130, 95, 115, 62,  
104, 138, 96, 109, 71, 124, 211, 165, 103, 202, 99, 21, 193, 4, 226,  
84, 229, 254, 106, 144, 241, 39, 86, 148, 132, 160, 104, 88, 232,  
228, 109, 85, 7, 86, 80, 134, 106, 166, 24, 92, 199, 210, 188, 153,  
187, 218, 69, 227]
```

The resulting "authentication tag" value is:

```
[154, 35, 80, 107, 37, 148, 81, 6, 103, 4, 60, 206, 171, 165, 113,  
67]
```

A.3.9. Encoded JWE Ciphertext

Base64url encode the resulting Ciphertext to create the Encoded JWE Ciphertext. This result (with line breaks for display purposes only) is:

```
4wxZhLkQ-F2RVzWCX3M-aIpgbUd806VnymMVwQTiVOX-apDxJlaUhKBoWOjkbVUH  
VlCGaqYYXMfSvJm72kXj
```

A.3.10. Encoded JWE Integrity Value

Base64url encode the resulting "authentication tag" to create the Encoded JWE Integrity Value. This result is:

```
miNQayWUUQZnBDzOq6VxQw
```

A.3.11. Complete Representation

Assemble the final representation: The Compact Serialization of this result is the concatenation of the Encoded JWE Header, the Encoded JWE Encrypted Key, the Encoded JWE Initialization Vector, the Encoded JWE Ciphertext, and the Encoded JWE Integrity Value in that order, with the five strings being separated by four period ('.') characters.

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

```
eyJhbGciOiJBMTI4S1ciLCJlbmMiOiJBMTI4R0NNIn0.  
pP_7AUDIQcgixVGPK9PwJr-htXV3RCxQ.  
_dxQGaaYsqhhY0NZ.  
4wxZhLkQ-F2RVzWCX3M-aIpgbUd806VnymMVwQTiVOX-apDxJlaUhKBoWOjkbVUH  
VlCGaqYYXMfSvJm72kXj.  
miNQayWUUQZnBDzOq6VxQw
```

A.3.12. Validation

This example illustrates the process of creating a JWE with symmetric key wrap and an AEAD algorithm. 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 Key Derivation for "enc" value "A128CBC+HS256"

This example uses the Concat KDF to derive the Content Encryption Key (CEK) and Content Integrity Key (CIK) from the Content Master Key (CMK) in the manner described in Section 4.8.1 of [JWA]. In this example, a 256 bit CMK is used to derive a 128 bit CEK and a 256 bit CIK.

The CMK value used 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.4.1. CEK Generation

These values are concatenated to produce the round 1 hash input:

- o the round number 1 as a 32 bit big endian integer ([0, 0, 0, 1]),
- o the CMK value (as above),
- o the output bit size 128 as a 32 bit big endian number ([0, 0, 0, 128]),
- o the bytes of the UTF-8 representation of the "enc" value "A128CBC+HS256" -- [65, 49, 50, 56, 67, 66, 67, 43, 72, 83, 50, 53, 54],
- o (no bytes are included for the "epu" (encryption PartyUInfo) and "epv" (encryption PartyVInfo) parameters because they are absent, but if present, the base64url decoded values of them would have been included here),
- o the bytes of the ASCII representation of the label "Encryption" -- [69, 110, 99, 114, 121, 112, 116, 105, 111, 110].

Thus the round 1 hash input is:

[0, 0, 0, 1, 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, 0, 0, 0, 128, 65, 49, 50, 56, 67, 66, 67, 43, 72, 83, 50, 53, 54, 69, 110, 99, 114, 121, 112, 116, 105, 111, 110]

The SHA-256 hash of this value, which is the round 1 hash output, is:

[37, 245, 125, 247, 113, 155, 238, 98, 228, 206, 62, 65, 81, 153, 79, 91, 225, 37, 250, 101, 198, 63, 51, 182, 5, 242, 241, 169, 162, 232, 103, 155]

Given that 128 bits are needed for the CEK and the hash has produced 256 bits, the CEK value is the first 128 bits of that value:

[37, 245, 125, 247, 113, 155, 238, 98, 228, 206, 62, 65, 81, 153, 79, 91]

A.4.2. CIK Generation

These values are concatenated to produce the round 1 hash input:

- o the round number 1 as a 32 bit big endian integer ([0, 0, 0, 1]),
- o the CMK value (as above),
- o the output bit size 256 as a 32 bit big endian number ([0, 0, 1, 0]),
- o the bytes of the UTF-8 representation of the "enc" value "A128CBC+HS256" -- [65, 49, 50, 56, 67, 66, 67, 43, 72, 83, 50, 53, 54],
- o (no bytes are included for the "epu" (encryption PartyUInfo) and "epv" (encryption PartyVInfo) parameters because they are absent, but if present, the base64url decoded values of them would have been included here),
- o the bytes of the ASCII representation of the label "Integrity" -- [73, 110, 116, 101, 103, 114, 105, 116, 121].

Thus the round 1 hash input is:

```
[0, 0, 0, 1, 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, 0, 0, 1, 0, 65, 49, 50, 56, 67, 66, 67, 43,
72, 83, 50, 53, 54, 73, 110, 116, 101, 103, 114, 105, 116, 121]
```

The SHA-256 hash of this value, which is the round 1 hash output, is:

```
[203, 194, 197, 180, 120, 46, 123, 202, 78, 12, 33, 116, 214, 247,
128, 41, 175, 53, 181, 164, 224, 223, 56, 146, 179, 193, 18, 223,
146, 85, 244, 127]
```

Given that 256 bits are needed for the CIK and the hash has produced 256 bits, the CIK value is that same value:

```
[203, 194, 197, 180, 120, 46, 123, 202, 78, 12, 33, 116, 214, 247,
128, 41, 175, 53, 181, 164, 224, 223, 56, 146, 179, 193, 18, 223,
146, 85, 244, 127]
```

A.5. Example Key Derivation for "enc" value "A256CBC+HS512"

This example uses the Concat KDF to derive the Content Encryption Key (CEK) and Content Integrity Key (CIK) from the Content Master Key (CMK) in the manner described in Section 4.8.1 of [JWA]. In this

example, a 512 bit CMK is used to derive a 256 bit CEK and a 512 bit CIK.

The CMK value used is:

```
[148, 116, 199, 126, 2, 117, 233, 76, 150, 149, 89, 193, 61, 34, 239,
226, 109, 71, 59, 160, 192, 140, 150, 235, 106, 204, 49, 176, 68,
119, 13, 34, 49, 19, 41, 69, 5, 20, 252, 145, 104, 129, 137, 138, 67,
23, 153, 83, 81, 234, 82, 247, 48, 211, 41, 130, 35, 124, 45, 156,
249, 7, 225, 168]
```

A.5.1. CEK Generation

These values are concatenated to produce the round 1 hash input:

- o the round number 1 as a 32 bit big endian integer ([0, 0, 0, 1]),
- o the CMK value (as above),
- o the output bit size 256 as a 32 bit big endian number ([0, 0, 1, 0]),
- o the bytes of the UTF-8 representation of the "enc" value "A256CBC+HS512" -- [65, 50, 53, 54, 67, 66, 67, 43, 72, 83, 53, 49, 50],
- o (no bytes are included for the "epu" (encryption PartyUInfo) and "epv" (encryption PartyVInfo) parameters because they are absent, but if present, the base64url decoded values of them would have been included here),
- o the bytes of the ASCII representation of the label "Encryption" -- [69, 110, 99, 114, 121, 112, 116, 105, 111, 110].

Thus the round 1 hash input is:

```
[0, 0, 0, 1, 148, 116, 199, 126, 2, 117, 233, 76, 150, 149, 89, 193,
61, 34, 239, 226, 109, 71, 59, 160, 192, 140, 150, 235, 106, 204, 49,
176, 68, 119, 13, 34, 49, 19, 41, 69, 5, 20, 252, 145, 104, 129, 137,
138, 67, 23, 153, 83, 81, 234, 82, 247, 48, 211, 41, 130, 35, 124,
45, 156, 249, 7, 225, 168, 0, 0, 1, 0, 65, 50, 53, 54, 67, 66, 67,
43, 72, 83, 53, 49, 50, 69, 110, 99, 114, 121, 112, 116, 105, 111,
110]
```

The SHA-512 hash of this value, which is the round 1 hash output, is:

```
[95, 112, 19, 252, 0, 97, 200, 188, 108, 84, 27, 116, 192, 169, 42,
165, 25, 246, 115, 235, 226, 198, 148, 211, 94, 143, 240, 226, 89,
```

226, 79, 13, 178, 80, 124, 251, 55, 114, 30, 115, 179, 64, 107, 213,
222, 225, 12, 169, 245, 116, 231, 83, 227, 233, 20, 164, 249, 148,
62, 92, 43, 5, 1, 97]

Given that 256 bits are needed for the CEK and the hash has produced 512 bits, the CEK value is the first 256 bits of that value:

[95, 112, 19, 252, 0, 97, 200, 188, 108, 84, 27, 116, 192, 169, 42,
165, 25, 246, 115, 235, 226, 198, 148, 211, 94, 143, 240, 226, 89,
226, 79, 13]

A.5.2. CIK Generation

These values are concatenated to produce the round 1 hash input:

- o the round number 1 as a 32 bit big endian integer ([0, 0, 0, 1]),
- o the CMK value (as above),
- o the output bit size 512 as a 32 bit big endian number ([0, 0, 2, 0]),
- o the bytes of the UTF-8 representation of the "enc" value "A256CBC+HS512" -- [65, 50, 53, 54, 67, 66, 67, 43, 72, 83, 53, 49, 50],
- o (no bytes are included for the "epu" (encryption PartyUInfo) and "epv" (encryption PartyVInfo) parameters because they are absent, but if present, the base64url decoded values of them would have been included here),
- o the bytes of the ASCII representation of the label "Integrity" -- [73, 110, 116, 101, 103, 114, 105, 116, 121].

Thus the round 1 hash input is:

[0, 0, 0, 1, 148, 116, 199, 126, 2, 117, 233, 76, 150, 149, 89, 193,
61, 34, 239, 226, 109, 71, 59, 160, 192, 140, 150, 235, 106, 204, 49,
176, 68, 119, 13, 34, 49, 19, 41, 69, 5, 20, 252, 145, 104, 129, 137,
138, 67, 23, 153, 83, 81, 234, 82, 247, 48, 211, 41, 130, 35, 124,
45, 156, 249, 7, 225, 168, 0, 0, 2, 0, 65, 50, 53, 54, 67, 66, 67,
43, 72, 83, 53, 49, 50, 73, 110, 116, 101, 103, 114, 105, 116, 121]

The SHA-512 hash of this value, which is the round 1 hash output, is:

[203, 188, 104, 71, 177, 60, 21, 10, 255, 157, 56, 214, 254, 87, 32,
115, 194, 36, 117, 162, 226, 93, 50, 220, 191, 219, 41, 56, 80, 197,
18, 173, 250, 145, 215, 178, 235, 51, 251, 122, 212, 193, 48, 227,

126, 89, 253, 101, 143, 252, 124, 157, 147, 200, 175, 164, 253, 92,
204, 122, 218, 77, 105, 146]

Given that 512 bits are needed for the CIK and the hash has produced 512 bits, the CIK value is that same value:

[203, 188, 104, 71, 177, 60, 21, 10, 255, 157, 56, 214, 254, 87, 32,
115, 194, 36, 117, 162, 226, 93, 50, 220, 191, 219, 41, 56, 80, 197,
18, 173, 250, 145, 215, 178, 235, 51, 251, 122, 212, 193, 48, 227,
126, 89, 253, 101, 143, 252, 124, 157, 147, 200, 175, 164, 253, 92,
204, 122, 218, 77, 105, 146]

Appendix B. Acknowledgements

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

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

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

Jim Schaad and Karen O'Donoghue chaired the JOSE working group and Sean Turner and Stephen Farrell served as Security area directors during the creation of this specification.

Appendix C. Open Issues

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

The following items remain to be considered or done in this draft:

- o Should we define optional nonce, timestamp, and/or uninterpreted string header parameter(s)?

Appendix D. Document History

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

-06

- o Removed the "int" and "kdf" parameters and defined the new composite AEAD algorithms "A128CBC+HS256" and "A256CBC+HS512" to replace the former uses of AES CBC, which required the use of separate integrity and key derivation functions.
- o Included additional values in the Concat KDF calculation -- the desired output size and the algorithm value, and optionally PartyUInfo and PartyVInfo values. Added the optional header parameters "apu" (agreement PartyUInfo), "apv" (agreement PartyVInfo), "epu" (encryption PartyUInfo), and "epv" (encryption PartyVInfo). Updated the KDF examples accordingly.
- o Promoted Initialization Vector from being a header parameter to being a top-level JWE element. This saves approximately 16 bytes in the compact serialization, which is a significant savings for some use cases. Promoting the Initialization Vector out of the header also avoids repeating this shared value in the JSON serialization.
- o Changed "x5c" (X.509 Certificate Chain) representation from being a single string to being an array of strings, each containing a single base64 encoded DER certificate value, representing elements of the certificate chain.
- o Added an AES Key Wrap example.
- o Reordered the encryption steps so CMK creation is first, when required.
- o Correct statements in examples about which algorithms produce reproducible results.

-05

- o Support both direct encryption using a shared or agreed upon symmetric key, and the use of a shared or agreed upon symmetric key to key wrap the CMK.
- o Added statement that "StringOrURI values are compared as case-sensitive strings with no transformations or canonicalizations applied".

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

-04

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

-03

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

- o Added Registry Contents sections to populate registry values.
- o Numerous editorial improvements.

-02

- o When using AEAD 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 Integrity Value.
- o Defined KDF output key sizes.
- o Generalized text to allow key agreement to be employed as an alternative to key wrapping or key encryption.
- o Changed compression algorithm from gzip to DEFLATE.
- o Clarified that it is an error when a "kid" value is included and no matching key is found.
- o Clarified that JWEs with duplicate Header Parameter Names MUST be rejected.
- o Clarified the relationship between "typ" header parameter values and MIME types.
- o Registered application/jwe MIME type and "JWE" typ header parameter value.
- o Simplified JWK terminology to get replace the "JWK Key Object" and "JWK Container Object" terms with simply "JSON Web Key (JWK)" and "JSON Web Key Set (JWK Set)" and to eliminate potential confusion between single keys and sets of keys. As part of this change, the header parameter name for a public key value was changed from "jpk" (JSON Public Key) to "jwk" (JSON Web Key).
- o Added suggestion on defining additional header parameters such as "x5t#S256" in the future for certificate thumbprints using hash algorithms other than SHA-1.
- o Specify RFC 2818 server identity validation, rather than RFC 6125 (paralleling the same decision in the OAuth specs).
- o Generalized language to refer to Message Authentication Codes (MACs) rather than Hash-based Message Authentication Codes (HMACs) unless in a context specific to HMAC algorithms.

- o Reformatted to give each header parameter its own section heading.
- 01
- o Added an integrity check for non-AEAD algorithms.
 - o Added "jpk" and "x5c" header parameters for including JWK public keys and X.509 certificate chains directly in the header.
 - o Clarified that this specification is defining the JWE Compact Serialization. Referenced the new JWE-JS spec, which defines the JWE JSON Serialization.
 - o Added text "New header parameters should be introduced sparingly since an implementation that does not understand a parameter MUST reject the JWE".
 - o Clarified that the order of the encryption and decryption steps is not significant in cases where there are no dependencies between the inputs and outputs of the steps.
 - o Made other editorial improvements suggested by JOSE working group participants.

-00

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

Authors' Addresses

Michael B. Jones
Microsoft

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

Eric Rescorla
RTFM, Inc.

Email: ekr@rtfm.com

Joe Hildebrand
Cisco Systems, Inc.

Email: jhildebr@cisco.com

JOSE Working Group
Internet-Draft
Intended status: Standards Track
Expires: April 18, 2013

M. Jones
Microsoft
October 15, 2012

JSON Web Key (JWK)
draft-ietf-jose-json-web-key-06

Abstract

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

Status of this Memo

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

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

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

This Internet-Draft will expire on April 18, 2013.

Copyright Notice

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

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

described in the Simplified BSD License.

Table of Contents

1. Introduction	3
1.1. Notational Conventions	3
2. Terminology	3
3. Example JSON Web Key Set	4
4. JSON Web Key (JWK) Format	4
4.1. "alg" (Algorithm Family) Parameter	5
4.2. "use" (Key Use) Parameter	5
4.3. "kid" (Key ID) Parameter	5
5. JSON Web Key Set (JWK Set) Format	6
5.1. "keys" (JSON Web Key Set) Parameter	6
6. IANA Considerations	6
6.1. JSON Web Key Parameters Registry	7
6.1.1. Registration Template	7
6.1.2. Initial Registry Contents	7
6.2. JSON Web Key Set Parameters Registry	8
6.2.1. Registration Template	8
6.2.2. Initial Registry Contents	8
7. Security Considerations	8
8. References	9
8.1. Normative References	9
8.2. Informative References	9
Appendix A. Acknowledgements	10
Appendix B. Open Issues	10
Appendix C. Document History	10
Author's Address	12

1. Introduction

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

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

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

1.1. Notational Conventions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in Key words for use in RFCs to Indicate Requirement Levels [RFC2119].

2. Terminology

JSON Web Key (JWK) A JSON data structure that represents a public key.

JSON Web Key Set (JWK Set) A JSON object that contains an array of JWKs as a member.

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

Collision Resistant Namespace A namespace that allows names to be allocated in a manner such that they are highly unlikely to collide with other names. For instance, collision resistance can be achieved through administrative delegation of portions of the namespace or through use of collision-resistant name allocation functions. 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.

3. Example JSON Web Key Set

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. Both provide a Key ID for key matching purposes. In both cases, integers are represented using the base64url encoding of their big endian representations. (Long lines are broken for display purposes only.)

```
{ "keys":
  [
    { "alg": "EC",
      "crv": "P-256",
      "x": "MKBCTNICKUSDiillySs3526iDZ8AiTo7Tu6KPAqv7D4",
      "y": "4Et16SRW2YiLUrN5vfVvVHuhp7x8Px1tmWW1bbM4IFyM",
      "use": "enc",
      "kid": "1" },
    { "alg": "RSA",
      "mod": "0vx7agoebGcQSuuPiLJXZptN9nndrQmbxEps2aiAFbWhM78LhWx
4cbbfAAAtVT86zwu1RK7aPFFxuhDR1L6tSoc_BjECPEbWKRXjBZCiFV4n3oknjhMs
tn64tZ_2W-5JsGY4Hc5n9yBXArw193lqt7_RN5w6Cf0h4QyQ5v-65YGjQR0_FDW2
QvzqY368QQMicAtaSqzs8KJZgnYb9c7d0zgdAZHzu6qmQvRL5hajrnlN9lCbOpbI
SD08qNlYrkt-bFTWhAI4vMQFh6WeZu0fM4lFd2NcRwr3XPksINHaQ-G_xBniIqb
w0LsljF44-csFCur-kEgU8awapJzKnqDKgw",
      "xpo": "AQAB",
      "kid": "2011-04-29" }
  ]
}
```

4. JSON Web Key (JWK) Format

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

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

The member names within a JWK MUST be unique; objects with duplicate member names MUST be rejected.

Additional members MAY be present in the JWK. If present, they MUST be understood by implementations using them. Member names used for representing key parameters for different kinds of keys need not be distinct. Member names SHOULD either be registered in the IANA JSON Web Key Parameters registry Section 6.1 or be URIs that contain a Collision Resistant Namespace.

4.1. "alg" (Algorithm Family) Parameter

The "alg" (algorithm family) member identifies the cryptographic algorithm family used with the key. "alg" values SHOULD either be registered in the IANA JSON Web Key Algorithm Families registry [JWA] or be a URI that contains a Collision Resistant Namespace. The "alg" value is a case sensitive string. This member is REQUIRED.

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

Additional members used with these "alg" values can be found in the IANA JSON Web Key Parameters registry Section 6.1; the initial contents of this registry are the values defined in Sections 5.2 and 5.3 of the JSON Web Algorithms (JWA) [JWA] specification.

4.2. "use" (Key Use) Parameter

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

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

Other values MAY be used. The "use" value is a case sensitive string. This member is OPTIONAL.

4.3. "kid" (Key ID) Parameter

The "kid" (key ID) member can be used to match a specific key. This can be used, for instance, to choose among a set of keys within a JWK Set during key rollover. The interpretation of the "kid" value is unspecified. Key ID values within a JWK Set need not be unique. The "kid" value is a case sensitive string. This member is OPTIONAL.

When used with JWS or JWE, the "kid" value MAY be used to match a JWS or JWE "kid" header parameter value.

In some contexts, different keys using the same Key ID value might be present, with the keys being disambiguated using other information, such as the "alg" or "use" values. For example, imagine "kid" values like "Current", "Upcoming", and "Deprecated", used for key rollover guidance. One could apply a label to all keys where the classification fits. If there are multiple "Current" keys, then in this example, they might be differentiated either by having different "alg" or "use" values, or some combination of both. As one example, there might only be one current RSA signing key and one current Elliptic Curve signing key, but both would be "Current".

5. JSON Web Key Set (JWK Set) Format

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

The member names within a JWK Set MUST be unique; objects with duplicate member names MUST be rejected.

Additional members MAY be present in the JWK Set. If present, they MUST be understood by implementations using them. Parameters for representing additional properties of JWK Sets SHOULD either be registered in the IANA JSON Web Key Set Parameters registry Section 6.2 or be a URI that contains a Collision Resistant Namespace.

5.1. "keys" (JSON Web Key Set) Parameter

The value of the "keys" (JSON Web Key Set) member is an array of JSON Web Key (JWK) values. This member is REQUIRED.

6. IANA Considerations

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

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

Registration requests must be sent to the [TBD]@ietf.org mailing list for review and comment, with an appropriate subject (e.g., "Request for access token type: example"). [[Note to RFC-EDITOR: The name of the mailing list should be determined in consultation with the IESG and IANA. Suggested name: jose-reg-review.]]

Within the review period, the Designated Expert(s) will either approve or deny the registration request, communicating this decision to the review list and IANA. Denials should include an explanation and, if applicable, suggestions as to how to make the request successful.

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

6.1. JSON Web Key Parameters Registry

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

6.1.1. Registration Template

Parameter Name:

The name requested (e.g., "example"). This name is case sensitive. Names that match other registered names in a case insensitive manner SHOULD NOT be accepted.

Change Controller:

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

Specification Document(s):

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

6.1.2. Initial Registry Contents

- o Parameter Name: "alg"
- o Change Controller: IETF

- o Specification Document(s): Section 4.1 of [[this document]]
- o Parameter Name: "use"
- o Change Controller: IETF
- o Specification Document(s): Section 4.2 of [[this document]]
- o Parameter Name: "kid"
- o Change Controller: IETF
- o Specification Document(s): Section 4.3 of [[this document]]

6.2. JSON Web Key Set Parameters Registry

This specification establishes the IANA JSON Web Key Set Parameters registry for reserved JWK Set parameter names. The registry records the reserved parameter name and a reference to the specification that defines it. This specification registers the parameter names defined in Section 5.

6.2.1. Registration Template

Parameter Name:

The name requested (e.g., "example"). This name is case sensitive. Names that match other registered names in a case insensitive manner SHOULD NOT be accepted.

Change Controller:

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

Specification Document(s):

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

6.2.2. Initial Registry Contents

- o Parameter Name: "keys"
- o Change Controller: IETF
- o Specification Document(s): Section 5.1 of [[this document]]

7. Security Considerations

All of the security issues faced by any cryptographic application must be faced by a JWS/JWE/JWK agent. Among these issues are protecting the user's private key, preventing various attacks, and

helping the user avoid mistakes such as inadvertently encrypting a message for the wrong recipient. The entire list of security considerations is beyond the scope of this document, but some significant concerns are listed here.

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

Per Section 4.3, applications should not assume that "kid" values are unique within a JWK Set.

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

8. References

8.1. Normative References

- [JWA] Jones, M., "JSON Web Algorithms (JWA)", October 2012.
- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.
- [RFC4627] Crockford, D., "The application/json Media Type for JavaScript Object Notation (JSON)", RFC 4627, July 2006.
- [RFC4648] Josefsson, S., "The Base16, Base32, and Base64 Data Encodings", RFC 4648, October 2006.
- [RFC5226] Narten, T. and H. Alvestrand, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 5226, May 2008.
- [W3C.CR-xmlsig-core2-20120124] Roessler, T., Yiu, K., Solo, D., Reagle, J., Datta, P., Eastlake, D., Hirsch, F., and S. Cantor, "XML Signature Syntax and Processing Version 2.0", World Wide Web Consortium CR CR-xmlsig-core2-20120124, January 2012, <<http://www.w3.org/TR/2012/CR-xmlsig-core2-20120124>>.

8.2. Informative References

- [JWE] Jones, M., Rescorla, E., and J. Hildebrand, "JSON Web Encryption (JWE)", October 2012.

- [JWS] Jones, M., Bradley, J., and N. Sakimura, "JSON Web Signature (JWS)", October 2012.
- [MagicSignatures] Panzer (editor), J., Laurie, B., and D. Balfanz, "Magic Signatures", January 2011.
- [RFC4122] Leach, P., Mealling, M., and R. Salz, "A Universally Unique IDentifier (UUID) URN Namespace", RFC 4122, July 2005.

Appendix A. Acknowledgements

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

Jim Schaad and Karen O'Donoghue chaired the JOSE working group and Sean Turner and Stephen Farrell served as Security area directors during the creation of this specification.

Appendix B. Open Issues

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

The following items remain to be considered or done in this draft:

- o No known open issues.

Appendix C. Document History

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

-06

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

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

-05

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

-04

- o Refer to the registries as the primary sources of defined values and then secondarily reference the sections defining the initial contents of the registries.
- o Normatively reference XML DSIG 2.0 [W3C.CR-xmlsig-core2-20120124] for its security considerations.
- o Added this language to Registration Templates: "This name is case sensitive. Names that match other registered names in a case insensitive manner SHOULD NOT be accepted."
- o Described additional open issues.
- o Applied editorial suggestions.

-03

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

-02

- o Simplified JWK terminology to get replace the "JWK Key Object" and "JWK Container Object" terms with simply "JSON Web Key (JWK)" and "JSON Web Key Set (JWK Set)" and to eliminate potential confusion between single keys and sets of keys. As part of this change, the

top-level member name for a set of keys was changed from "jwk" to "keys".

- o Clarified that values with duplicate member names MUST be rejected.
- o Established JSON Web Key Set Parameters registry.
- o Explicitly listed non-goals in the introduction.
- o Moved algorithm-specific definitions from JWK to JWA.
- o Reformatted to give each member definition its own section heading.

-01

- o Corrected the Magic Signatures reference.

-00

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

Author's Address

Michael B. Jones
Microsoft

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

JOSE Working Group
Internet-Draft
Intended status: Standards Track
Expires: April 18, 2013

M. Jones
Microsoft
J. Bradley
Ping Identity
N. Sakimura
NRI
October 15, 2012

JSON Web Signature (JWS)
draft-ietf-jose-json-web-signature-06

Abstract

JSON Web Signature (JWS) is a means of representing content secured with digital signatures or Message Authentication Codes (MACs) using JavaScript Object Notation (JSON) data structures. Cryptographic algorithms and identifiers for use with this specification are described in the separate JSON Web Algorithms (JWA) specification. Related encryption capabilities are described in the separate JSON Web Encryption (JWE) specification.

Status of this Memo

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

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

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

This Internet-Draft will expire on April 18, 2013.

Copyright Notice

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

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

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

Table of Contents

1. Introduction	4
1.1. Notational Conventions	4
2. Terminology	4
3. JSON Web Signature (JWS) Overview	5
3.1. Example JWS	6
4. JWS Header	7
4.1. Reserved Header Parameter Names	7
4.1.1. "alg" (Algorithm) Header Parameter	7
4.1.2. "jku" (JWK Set URL) Header Parameter	8
4.1.3. "jwk" (JSON Web Key) Header Parameter	8
4.1.4. "x5u" (X.509 URL) Header Parameter	8
4.1.5. "x5t" (X.509 Certificate Thumbprint) Header Parameter	8
4.1.6. "x5c" (X.509 Certificate Chain) Header Parameter	9
4.1.7. "kid" (Key ID) Header Parameter	9
4.1.8. "typ" (Type) Header Parameter	9
4.1.9. "cty" (Content Type) Header Parameter	10
4.2. Public Header Parameter Names	10
4.3. Private Header Parameter Names	10
5. Rules for Creating and Validating a JWS	10
6. Securing JWSs with Cryptographic Algorithms	12
7. IANA Considerations	13
7.1. JSON Web Signature and Encryption Header Parameters Registry	13
7.1.1. Registration Template	13
7.1.2. Initial Registry Contents	14
7.2. JSON Web Signature and Encryption Type Values Registry	15
7.2.1. Registration Template	15
7.2.2. Initial Registry Contents	15
7.3. Media Type Registration	16
7.3.1. Registry Contents	16
8. Security Considerations	16
8.1. Cryptographic Security Considerations	16
8.2. JSON Security Considerations	17
8.3. Unicode Comparison Security Considerations	18
9. References	18
9.1. Normative References	18
9.2. Informative References	20
Appendix A. JWS Examples	20

A.1.	JWS using HMAC SHA-256	20
A.1.1.	Encoding	20
A.1.2.	Decoding	22
A.1.3.	Validating	22
A.2.	JWS using RSA SHA-256	23
A.2.1.	Encoding	23
A.2.2.	Decoding	26
A.2.3.	Validating	26
A.3.	JWS using ECDSA P-256 SHA-256	26
A.3.1.	Encoding	26
A.3.2.	Decoding	28
A.3.3.	Validating	28
A.4.	JWS using ECDSA P-521 SHA-512	29
A.4.1.	Encoding	29
A.4.2.	Decoding	31
A.4.3.	Validating	31
A.5.	Example Plaintext JWS	32
Appendix B.	"x5c" (X.509 Certificate Chain) Example	32
Appendix C.	Notes on implementing base64url encoding without padding	34
Appendix D.	Acknowledgements	35
Appendix E.	Open Issues	36
Appendix F.	Document History	36
Authors' Addresses	39

1. Introduction

JSON Web Signature (JWS) is a compact format for representing content secured with digital signatures or Message Authentication Codes (MACs) intended for space constrained environments such as HTTP Authorization headers and URI query parameters. It represents this content using JavaScript Object Notation (JSON) [RFC4627] based data structures. The JWS cryptographic mechanisms provide integrity protection for arbitrary sequences of bytes.

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

1.1. Notational Conventions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in Key words for use in RFCs to Indicate Requirement Levels [RFC2119].

2. Terminology

JSON Web Signature (JWS) A data structure cryptographically securing a JWS Header and a JWS Payload with a JWS Signature value.

JWS Header A string representing a JSON object that describes the digital signature or MAC operation applied to create the JWS Signature value.

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

JWS Signature A byte array containing the cryptographic material that secures the JWS Header and the JWS Payload.

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

Encoded JWS Header Base64url encoding of the bytes of the UTF-8 [RFC3629] representation of the JWS Header.

Encoded JWS Payload Base64url encoding of the JWS Payload.

Encoded JWS Signature Base64url encoding of the JWS Signature.

JWS Secured Input The concatenation of the Encoded JWS Header, a period ('.') character, and the Encoded JWS Payload.

Header Parameter Name The name of a member of the JSON object representing a JWS Header.

Header Parameter Value The value of a member of the JSON object representing a JWS Header.

JWS Compact Serialization A representation of the JWS as the concatenation of the Encoded JWS Header, the Encoded JWS Payload, and the Encoded JWS Signature in that order, with the three strings being separated by two period ('.') characters.

Collision Resistant Namespace A namespace that allows names to be allocated in a manner such that they are highly unlikely to collide with other names. For instance, collision resistance can be achieved through administrative delegation of portions of the namespace or through use of collision-resistant name allocation functions. Examples of Collision Resistant Namespaces include: Domain Names, Object Identifiers (OIDs) as defined in the ITU-T X.660 and X.670 Recommendation series, and Universally Unique Identifiers (UUIDs) [RFC4122]. When using an administratively delegated namespace, the definer of a name needs to take reasonable precautions to ensure they are in control of the portion of the namespace they use to define the name.

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

3. JSON Web Signature (JWS) Overview

JWS represents digitally signed or MACed content using JSON data structures and base64url encoding. The representation consists of three parts: the JWS Header, the JWS Payload, and the JWS Signature. In the Compact Serialization, the three parts are base64url-encoded for transmission, and represented as the concatenation of the encoded strings in that order, with the three strings being separated by two period ('.') characters. (A JSON Serialization for this information is defined in the separate JSON Web Signature JSON Serialization

(JWS-JS) [JWS-JS] specification.)

The JWS Header describes the signature or MAC method and parameters employed. The JWS Payload is the message content to be secured. The JWS Signature ensures the integrity of both the JWS Header and the JWS Payload.

3.1. Example JWS

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

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

Base64url encoding the bytes of the UTF-8 representation of the JWS Header yields this Encoded JWS Header value:

```
eyJ0eXAiOiJKV1QiLA0KICJhbGciOiJIUzI1NiJ9
```

The following is an example of a JSON object that can be used as a 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 }
```

Base64url encoding the bytes of the UTF-8 representation of the JSON object yields the following Encoded JWS Payload (with line breaks for display purposes only):

```
eyJpc3MiOiJqb2UiLA0KICJleHAiOiJ0eEzMDA4MTkzODAsDQogImh0dHA6Ly9leGFt  
cGx1LmNvbS9pc19yb290Ijp0cnVlfQ
```

Computing the HMAC of the bytes of the ASCII [USASCII] representation of the JWS Secured Input (the concatenation of the Encoded JWS Header, a period ('.') character, and the Encoded JWS Payload) with the HMAC SHA-256 algorithm using the key specified in Appendix A.1 and base64url encoding the result yields this Encoded JWS Signature value:

```
dBjftJeZ4CVP-mB92K27uhbUJU1p1r_wW1gFWFOEjXk
```

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

```
eyJ0eXAiOiJKV1QiLA0KICJhbGciOiJIUzI1NiJ9
.
eyJpc3MiOiJqb2UiLA0KICJleHAiOjEzMDA4MTkzODAsDQogImh0dHA6Ly9leGFt
cGx1LmNvbS9pc19yb290Ijp0cnVlfQ
.
dBjftJeZ4CVP-mB92K27uhbUJU1plr_wWl9FwFOEjXk
```

This computation is illustrated in more detail in Appendix A.1.

4. JWS Header

The members of the JSON object represented by the JWS Header describe the digital signature or MAC applied to the Encoded JWS Header and the Encoded JWS Payload and optionally additional properties of the JWS. The Header Parameter Names within this object **MUST** be unique; JWSs with duplicate Header Parameter Names **MUST** be rejected. Implementations **MUST** understand the entire contents of the header; otherwise, the JWS **MUST** be rejected.

There are three classes of Header Parameter Names: Reserved Header Parameter Names, Public Header Parameter Names, and Private Header Parameter Names.

4.1. Reserved Header Parameter Names

The following header parameter names are reserved with meanings as defined below. All the names are short because a core goal of JWSs is for the representations to be compact.

Additional reserved header parameter names **MAY** be defined via the IANA JSON Web Signature and Encryption Header Parameters registry Section 7.1. 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 algorithm specified by the "alg" value **MUST** be supported by the implementation and there **MUST** be a key for use with that algorithm associated with the party that digitally signed or MACed the content or the JWS **MUST** be rejected. "alg" values **SHOULD** either be registered in the IANA JSON Web Signature and Encryption Algorithms registry [JWA] or be a URI that contains a Collision Resistant Namespace. The "alg" value is a case sensitive string containing a StringOrURI value. This header

parameter is REQUIRED.

A list of defined "alg" values can be found in the IANA JSON Web Signature and Encryption Algorithms registry [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 certificate MUST use TLS [RFC2818] [RFC5246]; the identity of the server MUST be validated, as per Section 3.1 of HTTP Over TLS [RFC2818]. This header parameter is OPTIONAL.

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

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

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

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

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

The "x5t" (X.509 Certificate Thumbprint) header parameter provides 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. This header parameter is OPTIONAL.

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

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

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

See Appendix B for an example "x5c" value.

4.1.7. "kid" (Key ID) Header Parameter

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

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

4.1.8. "typ" (Type) Header Parameter

The "typ" (type) header parameter is used to declare the type of this object. The type value "JWS" MAY be used to indicate that this object is a JWS. The "typ" value is a case sensitive string. This header parameter is OPTIONAL.

MIME Media Type [RFC2046] values MAY be used as "typ" values.

"typ" values SHOULD either be registered in the IANA JSON Web Signature and Encryption Type Values registry Section 7.2 or be a URI that contains a Collision Resistant Namespace.

4.1.9. "cty" (Content Type) Header Parameter

The "cty" (content type) header parameter is used to declare the type of the secured content (the Payload). The "cty" value is a case sensitive string. This header parameter is OPTIONAL.

The values used for the "cty" header parameter come from the same value space as the "typ" header parameter, with the same rules applying.

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 Section 7.1 or be a URI that contains a Collision Resistant Namespace. 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 any header parameter name that is not a Reserved Name Section 4.1 or a Public Name Section 4.2. Unlike Public Names, these private names are subject to collision and should be used with caution.

5. Rules for Creating and Validating a JWS

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

1. Create the content to be used as the JWS Payload.
2. Base64url encode the bytes of the JWS Payload. This encoding becomes the Encoded JWS Payload.

3. Create a JWS Header containing the desired set of header parameters. Note that white space is explicitly allowed in the representation and no canonicalization need be performed before encoding.
4. Base64url encode the bytes of the UTF-8 representation of the JWS Header to create the Encoded JWS Header.
5. Compute the JWS Signature in the manner defined for the particular algorithm being used. The JWS Secured Input is always the concatenation of the Encoded JWS Header, a period ('.') character, and the Encoded JWS Payload. The "alg" (algorithm) header parameter MUST be present in the JSON Header, with the algorithm value accurately representing the algorithm used to construct the JWS Signature.
6. Base64url encode the representation of the JWS Signature to create the Encoded JWS Signature.
7. The three encoded parts, taken together, are the result. The Compact Serialization of this result is the concatenation of the Encoded JWS Header, the Encoded JWS Payload, and the Encoded JWS Signature in that order, with the three strings being separated by two period ('.') characters.

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

1. Parse the three parts of the input (which are separated by period ('.') characters when using the JWS Compact Serialization) into the Encoded JWS Header, the Encoded JWS Payload, and the Encoded JWS Signature.
2. The Encoded JWS Header MUST be successfully base64url decoded following the restriction given in this specification that no padding characters have been used.
3. The resulting JWS Header MUST be completely valid JSON syntax conforming to RFC 4627 [RFC4627].
4. The resulting JWS Header MUST be validated to only include parameters and values whose syntax and semantics are both understood and supported.
5. The Encoded JWS Payload MUST be successfully base64url decoded following the restriction given in this specification that no

padding characters have been used.

6. The Encoded JWS Signature MUST be successfully base64url decoded following the restriction given in this specification that no padding characters have been used.
7. The JWS Signature MUST be successfully validated against the JWS Secured Input (the concatenation of the Encoded JWS Header, a period ('.') character, and the Encoded 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.

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

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

1. Remove any JSON applied escaping to produce an array of Unicode code points.
2. Unicode Normalization [USA15] MUST NOT be applied at any point to either the JSON string or to the string it is to be compared against.
3. Comparisons between the two strings MUST be performed as a Unicode code point to code point equality comparison.

6. Securing JWSs with Cryptographic Algorithms

JWS uses cryptographic algorithms to digitally sign or MAC the JWS Header and the JWS Payload. The JSON Web Algorithms (JWA) [JWA] specification describes a set of cryptographic algorithms and identifiers to be used with this specification. Specifically, Section 3.1 specifies a set of "alg" (algorithm) header parameter values intended for use this specification. It also describes the semantics and operations that are specific to these algorithms and algorithm families.

Public keys employed for digital signing can be identified using the Header Parameter methods described in Section 4.1 or can be distributed using methods that are outside the scope of this

specification.

7. IANA Considerations

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

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

Registration requests must be sent to the [TBD]@ietf.org mailing list for review and comment, with an appropriate subject (e.g., "Request for access token type: example"). [[Note to RFC-EDITOR: The name of the mailing list should be determined in consultation with the IESG and IANA. Suggested name: jose-reg-review.]]

Within the review period, the Designated Expert(s) will either approve or deny the registration request, communicating this decision to the review list and IANA. Denials should include an explanation and, if applicable, suggestions as to how to make the request successful.

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

7.1. JSON Web Signature and Encryption Header Parameters Registry

This specification establishes the IANA JSON Web Signature and Encryption Header Parameters registry for reserved JWS and JWE header parameter names. The registry records the reserved header parameter name and a reference to the specification that defines it. The same Header Parameter Name may be registered multiple times, provided that the parameter usage is compatible between the specifications.

7.1.1. Registration Template

Header Parameter Name:

The name requested (e.g., "example"). This name is case sensitive. Names that match other registered names in a case insensitive manner SHOULD NOT be accepted.

Change Controller:

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

Specification Document(s):

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

7.1.2. Initial Registry Contents

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

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

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

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

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

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

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

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

- o Header Parameter Name: "typ"
- o Change Controller: IETF

- o Specification Document(s): Section 4.1.8 of [[this document]]
- o Header Parameter Name: "cty"
- o Change Controller: IETF
- o Specification Document(s): Section 4.1.9 of [[this document]]

7.2. JSON Web Signature and Encryption Type Values Registry

This specification establishes the IANA JSON Web Signature and Encryption Type Values registry for values of the JWS and JWE "typ" (type) header parameter. It is RECOMMENDED that all registered "typ" values also include a MIME Media Type [RFC2046] value that the registered value is a short name for. The registry records the "typ" value, the MIME type value that it is an abbreviation for (if any), and a reference to the specification that defines it.

MIME Media Type [RFC2046] values MUST NOT be directly registered as new "typ" values; rather, new "typ" values MAY be registered as short names for MIME types.

7.2.1. Registration Template

"typ" Header Parameter Value:

The name requested (e.g., "example"). This name is case sensitive. Names that match other registered names in a case insensitive manner SHOULD NOT be accepted.

Abbreviation for MIME Type:

The MIME type that this name is an abbreviation for (e.g., "application/example").

Change Controller:

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

Specification Document(s):

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

7.2.2. Initial Registry Contents

This specification registers the "JWS" type value in this registry:

- o "typ" Header Parameter Value: "JWS"
- o Abbreviation for MIME type: application/jws
- o Change Controller: IETF
- o Specification Document(s): Section 4.1.8 of [[this document]]

7.3. Media Type Registration

7.3.1. Registry Contents

This specification registers the "application/jws" Media Type [RFC2046] in the MIME Media Type registry [RFC4288] to indicate that the content is a JWS using the Compact Serialization.

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

8. Security Considerations

8.1. Cryptographic Security Considerations

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

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

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

When utilizing TLS to retrieve information, the authority providing the resource MUST be authenticated and the information retrieved MUST be free from modification.

When cryptographic algorithms are implemented in such a way that successful operations take a different amount of time than unsuccessful operations, attackers may be able to use the time difference to obtain information about the keys employed. Therefore, such timing differences must be avoided.

A SHA-1 hash is used when computing "x5t" (x.509 certificate thumbprint) values, for compatibility reasons. Should an effective means of producing SHA-1 hash collisions be developed, and should an attacker wish to interfere with the use of a known certificate on a given system, this could be accomplished by creating another certificate whose SHA-1 hash value is the same and adding it to the certificate store used by the intended victim. A prerequisite to this attack succeeding is the attacker having write access to the intended victim's certificate store.

If, in the future, certificate thumbprints need to be computed using hash functions other than SHA-1, it is suggested that additional related header parameters be defined for that purpose. For example, it is suggested that a new "x5t#S256" (X.509 Certificate Thumbprint using SHA-256) header parameter could be defined and used.

8.2. JSON Security Considerations

Strict JSON validation is a security requirement. If malformed JSON is received, then the intent of the sender is impossible to reliably discern. Ambiguous and potentially exploitable situations could arise if the JSON parser used does not reject malformed JSON syntax.

Section 2.2 of the JavaScript Object Notation (JSON) specification [RFC4627] states "The names within an object SHOULD be unique",

whereas this specification states that "Header Parameter Names within this object MUST be unique; JWSs with duplicate Header Parameter Names MUST be rejected". Thus, this specification requires that the Section 2.2 "SHOULD" be treated as a "MUST". Ambiguous and potentially exploitable situations could arise if the JSON parser used does not enforce the uniqueness of member names.

8.3. Unicode Comparison Security Considerations

Header parameter names and algorithm names are Unicode strings. For security reasons, the representations of these names must be compared verbatim after performing any escape processing (as per RFC 4627 [RFC4627], Section 2.5). This means, for instance, that these JSON strings must compare as being equal ("sig", "\u0073ig"), whereas these must all compare as being not equal to the first set or to each other ("SIG", "Sig", "si\u0047").

JSON strings MAY contain characters outside the Unicode Basic Multilingual Plane. For instance, the G clef character (U+1D11E) may be represented in a JSON string as "\uD834\uDD1E". Ideally, JWS implementations SHOULD ensure that characters outside the Basic Multilingual Plane are preserved and compared correctly; alternatively, if this is not possible due to these characters exercising limitations present in the underlying JSON implementation, then input containing them MUST be rejected.

9. References

9.1. Normative References

- [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)", October 2012.
- [JWK] Jones, M., "JSON Web Key (JWK)", October 2012.
- [RFC1421] Linn, J., "Privacy Enhancement for Internet Electronic Mail: Part I: Message Encryption and Authentication Procedures", RFC 1421, February 1993.
- [RFC2046] Freed, N. and N. Borenstein, "Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types", RFC 2046,

November 1996.

- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.
- [RFC2818] Rescorla, E., "HTTP Over TLS", RFC 2818, May 2000.
- [RFC3629] Yergeau, F., "UTF-8, a transformation format of ISO 10646", STD 63, RFC 3629, November 2003.
- [RFC3986] Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax", STD 66, RFC 3986, January 2005.
- [RFC4288] Freed, N. and J. Klensin, "Media Type Specifications and Registration Procedures", BCP 13, RFC 4288, December 2005.
- [RFC4627] Crockford, D., "The application/json Media Type for JavaScript Object Notation (JSON)", RFC 4627, July 2006.
- [RFC4648] Josefsson, S., "The Base16, Base32, and Base64 Data Encodings", RFC 4648, October 2006.
- [RFC5226] Narten, T. and H. Alvestrand, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 5226, May 2008.
- [RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, August 2008.
- [RFC5280] Cooper, D., Santesson, S., Farrell, S., Boeyen, S., Housley, R., and W. Polk, "Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile", RFC 5280, May 2008.
- [USA15] Davis, M., Whistler, K., and M. Duerst, "Unicode Normalization Forms", Unicode Standard Annex 15, 09 2009.
- [USASCII] American National Standards Institute, "Coded Character Set -- 7-bit American Standard Code for Information Interchange", ANSI X3.4, 1986.
- [W3C.WD-xmlsig-bestpractices-20110809] Datta, P. and F. Hirsch, "XML Signature Best Practices", World Wide Web Consortium WD WD-xmlsig-bestpractices-20110809, August 2011, <<http://www.w3.org/TR/2011/WD-xmlsig-bestpractices-20110809>>.

9.2. Informative References

- [CanvasApp] Facebook, "Canvas Applications", 2010.
- [JSS] Bradley, J. and N. Sakimura (editor), "JSON Simple Sign", September 2010.
- [JWE] Jones, M., Rescorla, E., and J. Hildebrand, "JSON Web Encryption (JWE)", October 2012.
- [JWS-JS] Jones, M., Bradley, J., and N. Sakimura, "JSON Web Signature JSON Serialization (JWS-JS)", October 2012.
- [JWT] Jones, M., Bradley, J., and N. Sakimura, "JSON Web Token (JWT)", October 2012.
- [MagicSignatures] Panzer (editor), J., Laurie, B., and D. Balfanz, "Magic Signatures", January 2011.
- [RFC4122] Leach, P., Mealling, M., and R. Salz, "A Universally Unique Identifier (UUID) URN Namespace", RFC 4122, July 2005.
- [W3C.CR-xmlsig-core2-20120124] Roessler, T., Yiu, K., Solo, D., Reagle, J., Datta, P., Eastlake, D., Hirsch, F., and S. Cantor, "XML Signature Syntax and Processing Version 2.0", World Wide Web Consortium CR CR-xmlsig-core2-20120124, January 2012, <<http://www.w3.org/TR/2012/CR-xmlsig-core2-20120124>>.

Appendix A. JWS Examples

This section provides several examples of JWSs. While these examples all represent JSON Web Tokens (JWTs) [JWT], the payload can be any base64url encoded content.

A.1. JWS using HMAC SHA-256

A.1.1. Encoding

The following example JWS Header declares that the data structure is a JSON Web Token (JWT) [JWT] and the JWS Secured Input is secured using the HMAC SHA-256 algorithm.

```
{ "typ": "JWT",
```

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

The following byte array contains the UTF-8 representation of the JWS Header:

```
[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]
```

Base64url encoding these bytes yields this Encoded JWS Header value:

```
eyJ0eXAiOiJKVlQiLA0KICJhbGciOiJIUzI1NiJ9
```

The JWS Payload used in this example is the bytes of the UTF-8 representation of the JSON object below. (Note that the payload can be any base64url encoded sequence of bytes, and need not be a base64url encoded JSON object.)

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

The following byte array, which is the UTF-8 representation of the JSON object above, is the JWS Payload:

```
[123, 34, 105, 115, 115, 34, 58, 34, 106, 111, 101, 34, 44, 13, 10,
32, 34, 101, 120, 112, 34, 58, 49, 51, 48, 48, 56, 49, 57, 51, 56,
48, 44, 13, 10, 32, 34, 104, 116, 116, 112, 58, 47, 47, 101, 120, 97,
109, 112, 108, 101, 46, 99, 111, 109, 47, 105, 115, 95, 114, 111,
111, 116, 34, 58, 116, 114, 117, 101, 125]
```

Base64url encoding the above yields the Encoded JWS Payload value (with line breaks for display purposes only):

```
eyJpc3MiOiJqb2UiLA0KICJleHAiOiJlZzMDA4MTkzODAsDQogImh0dHA6Ly9leGFt
cGx1LmNvbS9pc19yb290Ijp0cnVlfQ
```

Concatenating the Encoded JWS Header, a period ('.') character, and the Encoded JWS Payload yields this JWS Secured Input value (with line breaks for display purposes only):

```
eyJ0eXAiOiJKVlQiLA0KICJhbGciOiJIUzI1NiJ9
.
eyJpc3MiOiJqb2UiLA0KICJleHAiOiJlZzMDA4MTkzODAsDQogImh0dHA6Ly9leGFt
cGx1LmNvbS9pc19yb290Ijp0cnVlfQ
```

The ASCII representation of the JWS Secured Input is the following byte array:


```
[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 key represented by the following byte array:

```
[3, 35, 53, 75, 43, 15, 165, 188, 131, 126, 6, 101, 119, 123, 166,
143, 90, 179, 40, 230, 240, 84, 201, 40, 169, 15, 132, 178, 210, 80,
46, 191, 211, 251, 90, 146, 210, 6, 71, 239, 150, 138, 180, 195, 119,
98, 61, 34, 61, 46, 33, 114, 5, 46, 79, 8, 192, 205, 154, 245, 103,
208, 128, 163]
```

Running the HMAC SHA-256 algorithm on the bytes of the ASCII representation of the JWS Secured Input with this key yields the following byte array:

```
[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]
```

Base64url encoding the above HMAC output yields the Encoded JWS Signature value:

```
dBjftJeZ4CVP-mB92K27uhbUJUlplr_wWlgFWFOEjXk
```

A.1.2. Decoding

Decoding the JWS requires base64url decoding the Encoded JWS Header, Encoded JWS Payload, and Encoded JWS Signature to produce the JWS Header, JWS Payload, and JWS Signature byte arrays. The byte array containing the UTF-8 representation of the JWS Header is decoded into the JWS Header string.

A.1.3. Validating

Next we validate the decoded results. Since the "alg" parameter in the header is "HS256", we validate the HMAC SHA-256 value contained in the JWS Signature. If any of the validation steps fail, the JWS MUST be rejected.

First, we validate that the JWS Header string is legal JSON.

To validate the HMAC value, we repeat the previous process of using the correct key and the ASCII representation of the JWS Secured Input as input to the HMAC SHA-256 function and then taking the output and determining if it matches the JWS Signature. If it matches exactly, the HMAC has been validated.

A.2. JWS using RSA SHA-256

A.2.1. Encoding

The JWS 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 Header used is:

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

The following byte array contains the UTF-8 representation of the JWS Header:

```
[123, 34, 97, 108, 103, 34, 58, 34, 82, 83, 50, 53, 54, 34, 125]
```

Base64url encoding these bytes yields this Encoded JWS Header value:

```
eyJhbGciOiJSUzI1NiJ9
```

The JWS Payload used in this example, which follows, is the same as in the previous example. Since the Encoded JWS Payload will therefore be the same, its computation is not repeated here.

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

Concatenating the Encoded JWS Header, a period ('.') character, and the Encoded JWS Payload yields this JWS Secured Input value (with line breaks for display purposes only):

```
eyJhbGciOiJSUzI1NiJ9  
.  
eyJpc3MiOiJqb2UiLA0KICJleHAiOjEzMDA4MTkzODAsDQogImh0dHA6Ly9leGft  
cGx1LmNvbS9pc19yb290Ijp0cnVlfQ
```

The ASCII representation of the JWS Secured Input is the following byte array:

```
[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]

The RSA key consists of a public part (Modulus, Exponent), and a Private Exponent. The values of the RSA key used in this example, presented as the byte arrays representing big endian integers are:

Parameter Name	Value
Modulus	[161, 248, 22, 10, 226, 227, 201, 180, 101, 206, 141, 45, 101, 98, 99, 54, 43, 146, 125, 190, 41, 225, 240, 36, 119, 252, 22, 37, 204, 144, 161, 54, 227, 139, 217, 52, 151, 197, 182, 234, 99, 221, 119, 17, 230, 124, 116, 41, 249, 86, 176, 251, 138, 143, 8, 154, 220, 75, 105, 137, 60, 193, 51, 63, 83, 237, 208, 25, 184, 119, 132, 37, 47, 236, 145, 79, 228, 133, 119, 105, 89, 75, 234, 66, 128, 211, 44, 15, 85, 191, 98, 148, 79, 19, 3, 150, 188, 110, 155, 223, 110, 189, 210, 189, 163, 103, 142, 236, 160, 198, 104, 247, 1, 179, 141, 191, 251, 56, 200, 52, 44, 226, 254, 109, 39, 250, 222, 74, 90, 72, 116, 151, 157, 212, 185, 207, 154, 222, 196, 199, 91, 5, 133, 44, 44, 15, 94, 248, 165, 193, 117, 3, 146, 249, 68, 232, 237, 100, 193, 16, 198, 182, 71, 96, 154, 164, 120, 58, 235, 156, 108, 154, 215, 85, 49, 48, 80, 99, 139, 131, 102, 92, 111, 111, 122, 130, 163, 150, 112, 42, 31, 100, 27, 130, 211, 235, 242, 57, 34, 25, 73, 31, 182, 134, 135, 44, 87, 22, 245, 10, 248, 53, 141, 154, 139, 157, 23, 195, 64, 114, 143, 127, 135, 216, 154, 24, 216, 252, 171, 103, 173, 132, 89, 12, 46, 207, 117, 147, 57, 54, 60, 7, 3, 77, 111, 96, 111, 158, 33, 224, 84, 86, 202, 229, 233, 161]
Exponent	[1, 0, 1]

Private Exponent	[18, 174, 113, 164, 105, 205, 10, 43, 195, 126, 82, 108, 69, 0, 87, 31, 29, 97, 117, 29, 100, 233, 73, 112, 123, 98, 89, 15, 157, 11, 165, 124, 150, 60, 64, 30, 63, 207, 47, 44, 211, 189, 236, 136, 229, 3, 191, 198, 67, 155, 11, 40, 200, 47, 125, 55, 151, 103, 31, 82, 19, 238, 216, 193, 90, 37, 216, 213, 206, 160, 2, 94, 227, 171, 46, 139, 127, 121, 33, 111, 198, 59, 234, 86, 39, 83, 180, 6, 68, 198, 161, 81, 39, 217, 178, 149, 69, 64, 160, 187, 225, 163, 5, 86, 152, 45, 78, 159, 222, 95, 100, 37, 241, 77, 75, 113, 52, 65, 181, 93, 199, 59, 155, 74, 237, 204, 146, 172, 227, 146, 126, 55, 245, 125, 12, 253, 94, 117, 129, 250, 81, 44, 143, 73, 97, 169, 235, 11, 128, 248, 168, 7, 70, 114, 138, 85, 255, 70, 71, 31, 52, 37, 6, 59, 157, 83, 100, 47, 94, 222, 30, 132, 214, 19, 8, 26, 250, 92, 34, 208, 81, 40, 91, 214, 59, 148, 59, 86, 93, 137, 138, 5, 104, 84, 19, 229, 60, 60, 108, 101, 37, 255, 31, 227, 78, 61, 220, 112, 240, 213, 100, 80, 253, 164, 139, 161, 46, 16, 78, 157, 235, 159, 184, 24, 129, 225, 196, 189, 242, 93, 146, 71, 244, 80, 200, 101, 146, 121, 104, 231, 115, 52, 244, 65, 79, 117, 167, 80, 225, 57, 84, 110, 58, 138, 115, 157]
------------------	---

The RSA private key (Modulus, Private Exponent) is then passed to the RSA signing function, which also takes the hash type, SHA-256, and the bytes of the ASCII representation of the JWS Secured Input as inputs. The result of the digital signature is a byte array, 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]

Base64url encoding the digital signature produces this value for the Encoded JWS Signature (with line breaks for display purposes only):

```
cC4hiUPoj9Eetdgtv3hF80EGrhuB__dzERat0XF9g2VtQgr9PJbu3XOiZj5RZmh7
AAuHIm4Bh-0Qc_lF5YKt_O8W2Fp5jujGbds9uJdbF9CUAr7tldnZcAcQjbKBYNX4
BAynRFdiuB--f_nZLgrnbyTyWz075vRK5h6xBArLIARNPvkSjtQBMHlb1L07Qe7K
0GarZRmB_eSN9383LcOLn6_dO--xil2jzDwusC-eOkHWESqtFZESc6BfI7noOPqv
hJlphCnvWh6IeYI2w9QOYEUipUTI8np6LbgGY9Fs98rqVt5AXLIhWkWywlVmtVrB
p0igcN_IoypGlUPQGe77Rw
```

A.2.2. Decoding

Decoding the JWS requires base64url decoding the Encoded JWS Header, Encoded JWS Payload, and Encoded JWS Signature to produce the JWS Header, JWS Payload, and JWS Signature byte arrays. The byte array containing the UTF-8 representation of the JWS Header is decoded into the JWS Header string.

A.2.3. Validating

Since the "alg" parameter in the header is "RS256", we validate the RSA SHA-256 digital signature contained in the JWS Signature. If any of the validation steps fail, the JWS MUST be rejected.

First, we validate that the JWS Header string is legal JSON.

Validating the JWS Signature is a little different from the previous example. First, we base64url decode the Encoded JWS Signature to produce a digital signature S to check. We then pass (n, e), S and the bytes of the ASCII representation of the JWS Secured Input to an RSA signature verifier that has been configured to use the SHA-256 hash function.

A.3. JWS using ECDSA P-256 SHA-256

A.3.1. Encoding

The JWS Header for this example differs from the previous example because a different algorithm is being used. The JWS Header used is:

```
{"alg":"ES256"}
```

The following byte array contains the UTF-8 representation of the JWS Header:

```
[123, 34, 97, 108, 103, 34, 58, 34, 69, 83, 50, 53, 54, 34, 125]
```

Base64url encoding these bytes yields this Encoded JWS Header value:

```
eyJhbGciOiJFUzI1NiJ9
```

The JWS Payload used in this example, which follows, is the same as in the previous examples. Since the Encoded JWS Payload will therefore be the same, its computation is not repeated here.

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

Concatenating the Encoded JWS Header, a period ('.') character, and the Encoded JWS Payload yields this JWS Secured Input value (with line breaks for display purposes only):

```
eyJhbGciOiJFUzI1NiJ9
.
eyJpc3MiOiJqb2UiLA0KICJleHAiOjEzMDA4MTkzODAsDQogImh0dHA6Ly9leGft
cGx1LmNvbS9pc19yb290Ijp0cnVlfQ
```

The ASCII representation of the JWS Secured Input is the following byte array:

```
[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]
```

The ECDSA key consists of a public part, the EC point (x, y), and a private part d. The values of the ECDSA key used in this example, presented as the byte arrays representing three 256 bit big endian integers are:

Parameter Name	Value
x	[127, 205, 206, 39, 112, 246, 196, 93, 65, 131, 203, 238, 111, 219, 75, 123, 88, 7, 51, 53, 123, 233, 239, 19, 186, 207, 110, 60, 123, 209, 84, 69]
y	[199, 241, 68, 205, 27, 189, 155, 126, 135, 44, 223, 237, 185, 238, 185, 244, 179, 105, 93, 110, 169, 11, 36, 173, 138, 70, 35, 40, 133, 136, 229, 173]

d	[142, 155, 16, 158, 113, 144, 152, 191, 152, 4, 135, 223, 31, 93, 119, 233, 203, 41, 96, 110, 190, 210, 38, 59, 95, 87, 194, 19, 223, 132, 244, 178]
---	--

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 bytes of the ASCII representation of the JWS Secured 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 byte arrays 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]

Concatenating the S array to the end of the R array and base64url encoding the result produces this value for the Encoded JWS Signature (with line breaks for display purposes only):

```
DtEhU3ljBEG8L38VWafUAqOyKAM6-Xx-F4GawxaepmXFCgfTjDxw5djxLa8ISlSA
pmWQxfKTUJqPP3-Kg6NUlQ
```

A.3.2. Decoding

Decoding the JWS requires base64url decoding the Encoded JWS Header, Encoded JWS Payload, and Encoded JWS Signature to produce the JWS Header, JWS Payload, and JWS Signature byte arrays. The byte array containing the UTF-8 representation of the JWS Header is decoded into the JWS Header string.

A.3.3. Validating

Since the "alg" parameter in the header is "ES256", we validate the ECDSA P-256 SHA-256 digital signature contained in the JWS Signature. If any of the validation steps fail, the JWS MUST be rejected.

First, we validate that the JWS Header string is legal JSON.

Validating the JWS Signature is a little different from the first

example. First, we base64url decode the Encoded JWS Signature as in the previous examples but we then need to split the 64 member byte array that must result into two 32 byte arrays, the first R and the second S. We then pass (x, y), (R, S) and the bytes of the ASCII representation of the JWS Secured Input to an ECDSA signature verifier that has been configured to use the P-256 curve with the SHA-256 hash function.

As explained in Section 3.4 of the JSON Web Algorithms (JWA) [JWA] specification, the use of the K value in ECDSA means that we cannot validate the correctness of the digital signature in the same way we validated the correctness of the HMAC. Instead, implementations MUST use an ECDSA validator to validate the digital signature.

A.4. JWS using ECDSA P-521 SHA-512

A.4.1. Encoding

The JWS Header for this example differs from the previous example because a different ECDSA curve and hash function are used. The JWS Header used is:

```
{"alg":"ES512"}
```

The following byte array contains the UTF-8 representation of the JWS Header:

```
[123, 34, 97, 108, 103, 34, 58, 34, 69, 83, 53, 49, 50, 34, 125]
```

Base64url encoding these bytes yields this Encoded JWS Header value:

```
eyJhbGciOiJFUzUxMiJ9
```

The JWS Payload used in this example, is the ASCII string "Payload". The representation of this string is the byte array:

```
[80, 97, 121, 108, 111, 97, 100]
```

Base64url encoding these bytes yields the Encoded JWS Payload value:

```
UGF5bG9hZA
```

Concatenating the Encoded JWS Header, a period ('.') character, and the Encoded JWS Payload yields this JWS Secured Input value:

```
eyJhbGciOiJFUzUxMiJ9.UGF5bG9hZA
```

The ASCII representation of the JWS Secured Input is the following

byte array:

```
[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]
```

The ECDSA key consists of a public part, the EC point (x, y), and a private part d. The values of the ECDSA key used in this example, presented as the byte arrays representing three 521 bit big endian integers are:

Parameter Name	Value
x	[1, 233, 41, 5, 15, 18, 79, 198, 188, 85, 199, 213, 57, 51, 101, 223, 157, 239, 74, 176, 194, 44, 178, 87, 152, 249, 52, 235, 4, 227, 198, 186, 227, 112, 26, 87, 167, 145, 14, 157, 129, 191, 54, 49, 89, 232, 235, 203, 21, 93, 99, 73, 244, 189, 182, 204, 248, 169, 76, 92, 89, 199, 170, 193, 1, 164]
y	[0, 52, 166, 68, 14, 55, 103, 80, 210, 55, 31, 209, 189, 194, 200, 243, 183, 29, 47, 78, 229, 234, 52, 50, 200, 21, 204, 163, 21, 96, 254, 93, 147, 135, 236, 119, 75, 85, 131, 134, 48, 229, 203, 191, 90, 140, 190, 10, 145, 221, 0, 100, 198, 153, 154, 31, 110, 110, 103, 250, 221, 237, 228, 200, 200, 246]
d	[1, 142, 105, 111, 176, 52, 80, 88, 129, 221, 17, 11, 72, 62, 184, 125, 50, 206, 73, 95, 227, 107, 55, 69, 237, 242, 216, 202, 228, 240, 242, 83, 159, 70, 21, 160, 233, 142, 171, 82, 179, 192, 197, 234, 196, 206, 7, 81, 133, 168, 231, 187, 71, 222, 172, 29, 29, 231, 123, 204, 246, 97, 53, 230, 61, 130]

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 bytes of the ASCII representation of the JWS Secured 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 byte arrays 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]

Concatenating the S array to the end of the R array and base64url encoding the result produces this value for the Encoded JWS Signature (with line breaks for display purposes only):

```
AdwMgeerwtHoh-l192l60hp9wAHZfVJbLfD_UxMi70cwnZOYaRIlKPWROc-mZZq
wqT2SI-KGDKB34X00aw_7XdtAG8GaSwFKdCAPZgoXD2YBJZCPEx3xKpRwcd008Kp
EHwJjyqOgzDO7iKvU8vcnwNrmxYbSW9ERBXukOXolLzeO_Jn
```

A.4.2. Decoding

Decoding the JWS requires base64url decoding the Encoded JWS Header, Encoded JWS Payload, and Encoded JWS Signature to produce the JWS Header, JWS Payload, and JWS Signature byte arrays. The byte array containing the UTF-8 representation of the JWS Header is decoded into the JWS Header string.

A.4.3. Validating

Since the "alg" parameter in the header is "ES512", we validate the ECDSA P-521 SHA-512 digital signature contained in the JWS Signature. If any of the validation steps fail, the JWS MUST be rejected.

First, we validate that the JWS Header string is legal JSON.

Validating the JWS Signature is similar to the previous example. First, we base64url decode the Encoded JWS Signature as in the previous examples but we then need to split the 132 member byte array that must result into two 66 byte arrays, the first R and the second S. We then pass (x, y), (R, S) and the bytes of the ASCII representation of the JWS Secured Input to an ECDSA signature

verifier that has been configured to use the P-521 curve with the SHA-512 hash function.

As explained in Section 3.4 of the JSON Web Algorithms (JWA) [JWA] specification, the use of the K value in ECDSA means that we cannot validate the correctness of the digital signature in the same way we validated the correctness of the HMAC. Instead, implementations MUST use an ECDSA validator to validate the digital signature.

A.5. Example Plaintext JWS

The following example JWS Header declares that the encoded object is a Plaintext JWS:

```
{"alg":"none"}
```

Base64url encoding the bytes of the UTF-8 representation of the JWS Header yields this Encoded JWS Header:

```
eyJhbGciOiJub251In0
```

The JWS Payload used in this example, which follows, is the same as in the previous examples. Since the Encoded JWS Payload will therefore be the same, its computation is not repeated here.

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

The Encoded JWS Signature is the empty string.

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

```
eyJhbGciOiJub251In0  
.  
eyJpc3MiOiJqb2UiLA0KICJleHAiOjEzMDA4MTkzODAsDQogImh0dHA6Ly9leGFt  
cGx1LmNvbS9pc19yb290Ijp0cnVlfQ  
.
```

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

The JSON array below is an example of a certificate chain that could be used as the value of an "x5c" (X.509 Certificate Chain) header parameter, per Section 4.1.6. Note that since these strings contain base64 encoded (not base64url encoded) values, they are allowed to


```
QEWdWYDVR0TAQH/BAUwAwEB/zAzBggrBgEFBQcBAQQnMCUwIwYIKwYBBQUHMAGG
F2h0dHA6Ly9vY3NwLmdvZGFkZHkuY29tMEQGA1UdHwQ9MDswOAA3oDWGM2h0dHA
6Ly9jZXXJ0aWZpY2F0ZXMuZ29kYWRkeS5jb20vcmlvbn3NpdG9yeS9yb290LmNybd
BLBgNVHSAERDBCMEAGBFUdIAAwODA2BggrBgEFBQcCARYqaHR0cDovL2N1cnRpZ
mljYXRlcY5nb2RhZGR5LmNvbS9yZXBvc2l0b3J5MA4GA1UdDwEB/wQEAwIBBjAN
BgkqhkiG9w0BAQUFAAOBgQC1QPmnHfbq/qQaQlpE9xXUhUaJwL6e4+PrxeNYiY+
SnleocSxIOYGyeR+sBjUZsE4OWBsUs5iB0QQeyAfJg594RAoYC5jcdnp1DQ1tgM
QLARzLrUc+cb53S8wGd9D0VmsfSxOaFIqII6hR8INMqzW/Rn453HWkrugp++85j
09VZw==" ,
"MIIC5zCCA1ACAQEwDQYJKoZIhvcNAQEFBQAwbgsxJDAiBgNVBACtG1ZhbG1DZXJ
0IFZhbG1kYXRpb24gTmV0d29yazEXMBUGA1UEChMOVmFsaUN1cnQsIEluYy4xNT
AzBgNVBAsTLFZhbG1DZXJ0IENsYXNzIDIGUG9saWN5IFZhbG1kYXRpb24gQXV0a
G9yaXR5MSEwHwYDVQQDExhodHRwOi8vd3d3LnZhbG1jZXXJ0LmNvbS8xIDAeBgkq
hkiG9w0BCQEWEluZm9AdmFsaWN1cnQuY29tMB4XDTE5MDYyNjAwMTk1NFoXDTE
5MDYyNjAwMTk1NFowbgsxJDAiBgNVBACtG1ZhbG1DZXJ0IFZhbG1kYXRpb24gTm
V0d29yazEXMBUGA1UEChMOVmFsaUN1cnQsIEluYy4xNTAzBgNVBAsTLFZhbG1DZ
XJ0IENsYXNzIDIGUG9saWN5IFZhbG1kYXRpb24gQXV0aG9yaXR5MSEwHwYDVQQD
ExhodHRwOi8vd3d3LnZhbG1jZXXJ0LmNvbS8xIDAeBgkqhkiG9w0BCQEWEluZm9
AdmFsaWN1cnQuY29tMIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQDOOnHK5a
vIWZJV16vYdA757tn2VUdZZUcOBVXc65g2PFxTXdMwzzjsvUGJ7SVCCSRrC16zf
N1SLUzm1NZ9WlmpZdRJEy0kTRxQb7XBhVQ7/nHk01xC+YDgkRoKWzk2Z/M/VXwb
P7RfZHM047QSV4dk+NoS/zcnwbNDu+97bi5p9wIDAQABMA0GCSqGSIb3DQEBBQU
AA4GBADt/UG9vUJSZSWI4OB9L+KXIPqeCgfYrx+jFzug6EILLGACOTb2oWH+heQ
Clu+mNr0HZDzTuIYEZODJJKPTEjlbVUjP9UNV+mWwD5MlM/Mtsq2azSiGM5bUMM
j4QssxsodyamEwCW/POuZ6lcg5Ktz885hZo+L7tdEy8W9ViH0Pd"]
```

Appendix C. Notes on implementing base64url encoding without padding

This appendix describes how to implement base64url encoding and decoding functions without padding based upon standard base64 encoding and decoding functions that do use padding.

To be concrete, example C# code implementing these functions is shown below. Similar code could be used in other languages.

```
static string base64urlencode(byte [] arg)
{
    string s = Convert.ToBase64String(arg); // Standard 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 byte sequence below encodes into the string below, which when decoded, reproduces the byte sequence.

3 236 255 224 193

A-z_4ME

Appendix D. 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. Dirk Balfanz, Yaron Y. Goland, John Panzer, and Paul Tarjan all made significant contributions to the design of this specification.

Thanks to Axel Nennker for his early implementation and feedback on the JWS and JWE specifications.

Jim Schaad and Karen O'Donoghue chaired the JOSE working group and Sean Turner and Stephen Farrell served as Security area directors during the creation of this specification.

Appendix E. Open Issues

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

The following items remain to be considered or done in this draft:

- o Should we define optional nonce, timestamp, and/or uninterpreted string header parameter(s)?

Appendix F. Document History

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

-06

- o Changed "x5c" (X.509 Certificate Chain) representation from being a single string to being an array of strings, each containing a single base64 encoded DER certificate value, representing elements of the certificate chain.
- o Applied changes made by the RFC Editor to RFC 6749's registry language to this specification.

-05

- o Added statement that "StringOrURI values are compared as case-sensitive strings with no transformations or canonicalizations applied".
- o Indented artwork elements to better distinguish them from the body text.

-04

- o Completed JSON Security Considerations section, including considerations about rejecting input with duplicate member names.
- o Completed security considerations on the use of a SHA-1 hash when computing "x5t" (x.509 certificate thumbprint) values.

- o Refer to the registries as the primary sources of defined values and then secondarily reference the sections defining the initial contents of the registries.
- o Normatively reference XML DSIG 2.0 [W3C.CR-xmlsig-core2-20120124] for its security considerations.
- o Added this language to Registration Templates: "This name is case sensitive. Names that match other registered names in a case insensitive manner SHOULD NOT be accepted."
- o Reference draft-jones-jose-jws-json-serialization instead of draft-jones-json-web-signature-json-serialization.
- o Described additional open issues.
- o Applied editorial suggestions.

-03

- o Added the "cty" (content type) header parameter for declaring type information about the secured content, as opposed to the "typ" (type) header parameter, which declares type information about this object.
- o Added "Collision Resistant Namespace" to the terminology section.
- o Reference ITU.X690.1994 for DER encoding.
- o Added an example JWS using ECDSA P-521 SHA-512. This has particular illustrative value because of the use of the 521 bit integers in the key and signature values. This is also an example in which the payload is not a base64url encoded JSON object.
- o Added an example "x5c" value.
- o No longer say "the UTF-8 representation of the JWS Secured Input (which is the same as the ASCII representation)". Just call it "the ASCII representation of the JWS Secured Input".
- o Added Registration Template sections for defined registries.
- o Added Registry Contents sections to populate registry values.
- o Changed name of the JSON Web Signature and Encryption "typ" Values registry to be the JSON Web Signature and Encryption Type Values registry, since it is used for more than just values of the "typ" parameter.

- o Moved registries JSON Web Signature and Encryption Header Parameters and JSON Web Signature and Encryption Type Values to the JWS specification.
- o Numerous editorial improvements.

-02

- o Clarified that it is an error when a "kid" value is included and no matching key is found.
- o Removed assumption that "kid" (key ID) can only refer to an asymmetric key.
- o Clarified that JWSs with duplicate Header Parameter Names MUST be rejected.
- o Clarified the relationship between "typ" header parameter values and MIME types.
- o Registered application/jws MIME type and "JWS" typ header parameter value.
- o Simplified JWK terminology to get replace the "JWK Key Object" and "JWK Container Object" terms with simply "JSON Web Key (JWK)" and "JSON Web Key Set (JWK Set)" and to eliminate potential confusion between single keys and sets of keys. As part of this change, the header parameter name for a public key value was changed from "jpk" (JSON Public Key) to "jwk" (JSON Web Key).
- o Added suggestion on defining additional header parameters such as "x5t#S256" in the future for certificate thumbprints using hash algorithms other than SHA-1.
- o Specify RFC 2818 server identity validation, rather than RFC 6125 (paralleling the same decision in the OAuth specs).
- o Generalized language to refer to Message Authentication Codes (MACs) rather than Hash-based Message Authentication Codes (HMACs) unless in a context specific to HMAC algorithms.
- o Reformatted to give each header parameter its own section heading.

-01

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

- o Added "jpk" and "x5c" header parameters for including JWK public keys and X.509 certificate chains directly in the header.
- o Clarified that this specification is defining the JWS Compact Serialization. Referenced the new JWS-JS spec, which defines the JWS JSON Serialization.
- o Added text "New header parameters should be introduced sparingly since an implementation that does not understand a parameter MUST reject the JWS".
- o Clarified that the order of the creation and validation steps is not significant in cases where there are no dependencies between the inputs and outputs of the steps.
- o Changed "no canonicalization is performed" to "no canonicalization need be performed".
- o Corrected the Magic Signatures reference.
- o Made other editorial improvements suggested by JOSE working group participants.

-00

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

Authors' Addresses

Michael B. Jones
Microsoft

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

John Bradley
Ping Identity

Email: ve7jtb@ve7jtb.com

Nat Sakimura
Nomura Research Institute
Email: n-sakimura@nri.co.jp

JOSE Working Group
Internet-Draft
Intended status: Standards Track
Expires: February 16, 2013

M. Jones
Microsoft
August 15, 2012

JSON Private Key
draft-jones-jose-json-private-key-00

Abstract

The JSON Private Key specification extends the JSON Web Key (JWK) and JSON Web Algorithms (JWA) specifications to define a JavaScript Object Notation (JSON) representation of private keys.

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 February 16, 2013.

Copyright Notice

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

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

Table of Contents

1. Introduction	3
1.1. Notational Conventions	3
2. Terminology	3
3. JWK Parameters for Private Keys	3
3.1. JWK Parameters for Elliptic Curve Private Keys	3
3.1.1. "d" (ECC Private Key) Parameter	3
3.2. JWK Parameters for RSA Private Keys	3
3.2.1. "pri" (Private Exponent) Parameter	4
4. Example Private Keys	4
5. IANA Considerations	5
5.1. JSON Web Key Parameters Registration	5
5.1.1. Registry Contents	5
6. Security Considerations	5
7. Normative References	5
Appendix A. Document History	5
Author's Address	6

1. Introduction

The JSON Private Key specification extends the JSON Web Key (JWK) [JWK] and JSON Web Algorithms (JWA) [JWA] specifications to define a JavaScript Object Notation (JSON) [RFC4627] representation of private keys.

1.1. Notational Conventions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in Key words for use in RFCs to Indicate Requirement Levels [RFC2119].

2. Terminology

This specification uses the same terminology as the JSON Web Key (JWK) [JWK] and JSON Web Algorithms (JWA) [JWA] specifications.

3. JWK Parameters for Private Keys

This section defines additional JSON Web Key parameters that enable JWKs to represent private keys.

3.1. JWK Parameters for Elliptic Curve Private Keys

When the JWK "alg" member value is "EC", the following member MAY be used to represent an Elliptic Curve private key:

3.1.1. "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 value's unsigned big endian representation as a byte array. The array representation MUST not be shortened to omit any leading zero bytes. For instance, when representing 521 bit integers, the byte array to be base64url encoded MUST contain 66 bytes, including any leading zero bytes.

3.2. JWK Parameters for RSA Private Keys

When the JWK "alg" member value is "RSA", the following member MAY be used to represent an RSA private key:

3.2.1. "pri" (Private Exponent) Parameter

The "pri" (private exponent) member contains the private exponent value for the RSA private key. It is represented as the base64url encoding of the value's unsigned big endian representation as a byte array. The array representation MUST not be shortened to omit any leading zero bytes. For instance, when representing 2048 bit integers, the byte array to be base64url encoded MUST contain 256 bytes, including any leading zero bytes.

4. 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 Section 3 of [JWK], adding private key values. (Line breaks are for display purposes only.)

```
{ "keys":
  [
    { "alg": "EC",
      "crv": "P-256",
      "x": "MKBCTNiCKUSDiillySs3526iDZ8AiTo7Tu6KPAqv7D4",
      "y": "4Et16SRW2YiLUrN5vfVvHuhp7x8Px1tmWWlbbM4IFyM",
      "d": "870MB6gfuTJ4HtUnUvYMyJpr5eUZNP4Bk43bVdj3eAE",
      "use": "enc",
      "kid": "1" },

    { "alg": "RSA",
      "mod": "0vx7agoebGcQSuuPiLJXZptN9nndrQmbXEps2aiAFbWhM78LhWx4
cbbfAAAtVT86zwu1RK7aPFFxuhDR1L6tSoc_BJECPEbWKRXjBZCiFV4n3oknjhMst
n64tZ_2W-5JsGY4Hc5n9yBXArw193lqt7_RN5w6Cf0h4QyQ5v-65YGjQR0_FDW2Q
vzqY368QQMicAtaSqzs8KJZgnYb9c7d0zgdAZHzu6QMqvRL5hajrnl9lCbOpbIS
D08qNLYrckt-bFTWhAI4vMQFh6WeZu0fM4lFd2NcRwr3XPksINHaQ-G_xBniIqbw
0LsljF44-csFCur-kEgU8awapJzKnqDKgw",
      "exp": "AQAB",
      "pri": "X4cTteJY_gn4FYPsXB8rdXix5vwsg1FLN5E3EaG6RJoVH-HLLKD9
M7dx5oo7GURknchnrRweUkC7hT5fJLM0WbFAKNLWY2vv7B6NqXSzUvxT0_YSfqi j
wp3RTz1BaCxWp4doFk5N2o8Gy_nHNKroADIkJ46pRUohsXywbReAdYaMwFs9tv8d
_cPVY3i07a3t8MN6TNwm0dSawm9v47UiCl3Sk5ZiG7xojPLu4sbg1U2jx4IBTNBz
nbJSzFHK66jT8bgkuqsk0GjskDJk19Z4qwjwbsnn4j2WBii3RL-Us2lGVkY8fkFz
me1z0HbIkfz0Y6mqnOYtqc0X4jfcKoAC8Q",
      "kid": "2011-04-29" }
  ]
}
```


5. IANA Considerations

5.1. JSON Web Key Parameters Registration

This specification registers the parameter names defined in Section 3.1 and Section 3.2 in the IANA JSON Web Key Parameters registry [JWK].

5.1.1. Registry Contents

- o Parameter Name: "d"
- o Change Controller: IETF
- o Specification Document(s): Section 3.1.1 of [[this document]]
- o Parameter Name: "pri"
- o Change Controller: IETF
- o Specification Document(s): Section 3.2.1 of [[this document]]

6. Security Considerations

The security considerations for this specification are the same as those for the JSON Web Key (JWK) [JWK] specification and the portion of the JSON Web Algorithms (JWA) [JWA] specification that pertains to key representations.

7. Normative References

- [JWA] Jones, M., "JSON Web Algorithms (JWA)", July 2012.
- [JWK] Jones, M., "JSON Web Key (JWK)", July 2012.
- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.
- [RFC4627] Crockford, D., "The application/json Media Type for JavaScript Object Notation (JSON)", RFC 4627, July 2006.

Appendix A. Document History

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

-00

- o Created draft-jones-jose-json-private-key to facilitate discussion of the question from the W3C WebCrypto WG to the IETF JOSE WG of whether JOSE plans to support a format for representing private keys.

Author's Address

Michael B. Jones
Microsoft

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

JOSE Working Group
Internet-Draft
Intended status: Standards Track
Expires: April 18, 2013

M. Jones
Microsoft
October 15, 2012

JSON Web Encryption JSON Serialization (JWE-JS)
draft-jones-jose-jwe-json-serialization-02

Abstract

The JSON Web Encryption JSON Serialization (JWE-JS) is a means of representing encrypted content using JavaScript Object Notation (JSON) data structures. This specification describes a means of representing secured content as a JSON data object (as opposed to the JWE specification, which uses a compact serialization with a URL-safe representation). It enables the same content to be encrypted to multiple parties (unlike JWE). Cryptographic algorithms and identifiers used with this specification are described in the separate JSON Web Algorithms (JWA) specification. The JSON Serialization for related digital signature and MAC functionality is described in the separate JSON Web Signature JSON Serialization (JWS-JS) specification.

Status of this Memo

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

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

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

This Internet-Draft will expire on April 18, 2013.

Copyright Notice

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

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents

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

Table of Contents

1. Introduction	3
1.1. Notational Conventions	3
2. Terminology	3
3. JSON Serialization	3
4. Example JWE-JS	5
5. IANA Considerations	6
6. Security Considerations	6
7. References	6
7.1. Normative References	6
7.2. Informative References	7
Appendix A. Acknowledgements	7
Appendix B. Open Issues	7
Appendix C. Document History	7
Author's Address	8

1. Introduction

The JSON Web Encryption JSON Serialization (JWE-JS) is a format for representing encrypted content as a JavaScript Object Notation (JSON) [RFC4627] object. It enables the same content to be encrypted to multiple parties (unlike JWE [JWE].) The encryption mechanisms are independent of the type of content being encrypted. Cryptographic algorithms and identifiers used with this specification are described in the separate JSON Web Algorithms (JWA) [JWA] specification. The JSON Serialization for related digital signature and MAC functionality is described in the separate JSON Web Signature JSON Serialization (JWS-JS) [JWS-JS] specification.

1.1. Notational Conventions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in Key words for use in RFCs to Indicate Requirement Levels [RFC2119].

2. Terminology

This specification uses the same terminology as the JSON Web Encryption (JWE) [JWE] specification.

3. JSON Serialization

The JSON Serialization represents encrypted content as a JSON object with a "recipients" member containing an array of per-recipient information, an "initialization_vector" member containing a shared Encoded JWE Initialization Vector value, and a "ciphertext" member containing a shared Encoded JWE Ciphertext value. Each member of the "recipients" array is a JSON object with a "header" member containing an Encoded JWE Header value, an "encrypted_key" member containing an Encoded JWE Encrypted Key value, and an "integrity_value" member containing an Encoded JWE Integrity Value value.

Unlike the compact serialization used by JWEs, content using the JSON Serialization MAY be encrypted to more than one recipient. Each recipient requires:

- o a JWE Header value specifying the cryptographic parameters used to encrypt the JWE Encrypted Key to that recipient and the parameters used to encrypt the plaintext to produce the JWE Ciphertext; this is represented as an Encoded JWE Header value in the "header" member of an object in the "recipients" array.

- o a JWE Encrypted Key value used to encrypt the ciphertext; this is represented as an Encoded JWE Encrypted Key value in the "encrypted_key" member of the same object in the "recipients" array.
- o a JWE Integrity Value that ensures the integrity of the Ciphertext and the parameters used to create it; this is represented as an Encoded JWE Integrity Value value in the "integrity_value" member of the same object in the "recipients" array.

Therefore, the syntax is:

```
{ "recipients": [
  { "header": "<header 1 contents>",
    "encrypted_key": "<encrypted key 1 contents>",
    "integrity_value": "<integrity value 1 contents>" },
  ...
  { "header": "<header N contents>",
    "encrypted_key": "<encrypted key N contents>",
    "integrity_value": "<integrity value N contents>" } ],
  "initialization_vector": "<initialization vector contents>",
  "ciphertext": "<ciphertext contents>"
}
```

The contents of the Encoded JWE Header, Encoded JWE Encrypted Key, Encoded JWE Initialization Vector, Encoded JWE Ciphertext, and Encoded JWE Integrity Value values are exactly as specified in JSON Web Encryption (JWE) [JWE]. They are interpreted and validated in the same manner, with each corresponding "header", "encrypted_key", and "integrity_value" value being created and validated together.

Each JWE Encrypted Key value and the corresponding JWE Integrity Value are computed using the parameters of the corresponding JWE Header value in the same manner described in the JWE specification. This has the desirable result that each Encoded JWE Encrypted Key value in the "recipients" array and each Encoded JWE Integrity Value in the same array element are identical to the values that would have been computed for the same parameters in a JWE, as is the shared JWE Ciphertext value.

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

4. Example JWE-JS

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

Two recipients are present in this example: the first using the RSAES-PKCS1-V1_5 algorithm to encrypt the Content Master Key (CMK) and the second using RSAES OAEP to encrypt the CMK. The Plaintext is encrypted using the AES CBC algorithm and the same block encryption parameters to produce the common JWE Ciphertext value. The two Decoded JWE Header Segments used are:

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

and:

```
{ "alg": "RSA-OAEP", "enc": "A128CBC+HS256" }
```

The keys used for the first recipient are the same as those in Appendix A.2 of [JWE], as is the plaintext used. The asymmetric encryption key used for the second recipient is the same as that used in Appendix A.1 of [JWE]; the block encryption keys and parameters for the second recipient are the same as those for the first recipient (which must be the case, since the initialization vector and ciphertext are shared).

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


```

{
  "recipients": [
    {
      "header": {
        "eyJhbGciOiJSU0ExXzUiLCJlbnMiOiJBMTI4Q0JDK0hTMjU2In0",
        "encrypted_key":
          "O6AqXqgVlJJ4c4lp5sXZd7bpGHaw6ARKHUEXQxD1cAW4-X1x0qtj_AN0mukqE
          Ol4Y6UOWJXIjY9-G1ELK-RQWrKH_StR-AM9H7GpKmSEji8QYOcmOjr-u9H1Lt
          _pBEieG802SxWz0rbFTXRcj4BWLxcpCtjUZ31AP-sc-L_eCZ5UN10aSRNqFsk
          uPkzRsFZRDJqSSJeVOyJ7pZCQ83fli19Vgi_3R7XMUqluQuuc7ZHOWixi47jX
          lBTlWRZ5iFxaS8G6J8wUrd4BKggAw3qX5XoIfXQVlQZE0VmKq_zQSIo5LnFKy
          owoORcdsEuNh9B9Mkyt0ZQE1G-jGdtHWjZSOA",
        "integrity_value":
          "RBGhYzE8_cZLHjJqqHuLhzbGwGL_wV3LDSUrcbkOiIA"
      },
      {
        "header": {
          "eyJhbGciOiJSU0EtT0FFUCIsImVuYyI6IkExMjhDQkMrSFMjNTYifQ",
          "encrypted_key":
            "myoFYZHERXG4gMVWl9UrFOCFIwvOUudYrxTsRsOt6maTc3W8G1FqGVOIBSZve
            BdZz2LqS42xta5OXEwLYaocObUxtfH9H8vMsjO-mBo7U9mp_PkS9PqVJMkeEe
            PLhzNLH0ecq7nYT6AFr5sSt4WMOPjSwHVQWtx43fZt4HvYaE_vgeSrxdi8KAb
            xblzK_-qcYT6H7cwOMZrT6SfcXgLXESuKpF0azSGQtUmo0MLICP0YPBecGLTo
            PiveOH2awKZx0FkzPwi4JmOIvnAJ_wVQQJDVELwO9SioF8o1CQRHGyZ9rzDrr
            GRkoYgm2jVz-x0BuFVQFa4ZNufudtiT8pQxKg",
          "integrity_value":
            "i45dXWFjRkK805VtjIw_8iqGqlr9qPV7ULDLbnNAC_Q"
        }
      },
      "initialization_vector":
        "AxY8DctDaGlsbGljb3RoZQ",
      "ciphertext":
        "leBWFgcrz40wC88cgv8rPgu3EfmClp4zT0kIxxfSF2zDJcQ-iEHk1jQM95xAdr5
        Z"
    }
  ]
}

```

5. IANA Considerations

This specification makes no requests of IANA.

6. Security Considerations

The security considerations for this specification are the same as those for the JSON Web Encryption (JWE) [JWE] specification.

7. References

7.1. Normative References

[JWA] Jones, M., "JSON Web Algorithms (JWA)", October 2012.

- [JWE] Jones, M., Rescorla, E., and J. Hildebrand, "JSON Web Encryption (JWE)", October 2012.
- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.
- [RFC4627] Crockford, D., "The application/json Media Type for JavaScript Object Notation (JSON)", RFC 4627, July 2006.

7.2. Informative References

- [I-D.rescorla-jsms] Rescorla, E. and J. Hildebrand, "JavaScript Message Security Format", draft-rescorla-jsms-00 (work in progress), March 2011.
- [JSE] Bradley, J. and N. Sakimura (editor), "JSON Simple Encryption", September 2010.
- [JWS-JS] Jones, M., Bradley, J., and N. Sakimura, "JSON Web Signature JSON Serialization (JWS-JS)", October 2012.

Appendix A. Acknowledgements

JSON serializations for encrypted content were previously explored by JSON Simple Encryption [JSE] and JavaScript Message Security Format [I-D.rescorla-jsms].

Appendix B. Open Issues

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

The following items remain to be considered or done in this draft:

- o Track changes that occur in the JWE spec.

Appendix C. Document History

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

-02

- o Changed to use an array of structures for per-recipient values, rather than a set of parallel arrays.

- o Promoted Initialization Vector from being a header parameter to being a top-level JWE element. This saves approximately 16 bytes in the compact serialization, which is a significant savings for some use cases. Promoting the Initialization Vector out of the header also avoids repeating this shared value in the JSON serialization.

-01

- o Added a complete JWE-JS example.
- o Generalized language to refer to Message Authentication Codes (MACs) rather than Hash-based Message Authentication Codes (HMACs).

-00

- o Renamed draft-jones-json-web-encryption-json-serialization to draft-jones-jose-jwe-json-serialization to have "jose" be in the document name so it can be included in the Related Documents list at <http://datatracker.ietf.org/wg/jose/>. No normative changes.

draft-jones-json-web-encryption-json-serialization-02

- o Updated examples to track updated algorithm properties in the JWA spec.
- o Tracked editorial changes made to the JWE spec.

draft-jones-json-web-encryption-json-serialization-01

- o Tracked changes between JOSE JWE draft -00 and -01, which added an integrity check for non-AEAD algorithms.

draft-jones-json-web-encryption-json-serialization-00

- o Created the initial version incorporating JOSE working group input and drawing from the JSON Serialization previously proposed in draft-jones-json-web-token-01.

Author's Address

Michael B. Jones
Microsoft

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

JOSE Working Group
Internet-Draft
Intended status: Standards Track
Expires: April 18, 2013

M. Jones
Microsoft
J. Bradley
independent
N. Sakimura
Nomura Research Institute
October 15, 2012

JSON Web Signature JSON Serialization (JWS-JS)
draft-jones-jose-jws-json-serialization-02

Abstract

The JSON Web Signature JSON Serialization (JWS-JS) is a means of representing content secured with digital signatures or Message Authentication Codes (MACs) using JavaScript Object Notation (JSON) data structures. This specification describes a means of representing secured content as a JSON data object (as opposed to the JWS specification, which uses a compact serialization with a URL-safe representation). It enables multiple digital signatures and/or MACs to be applied to the same content (unlike JWS). Cryptographic algorithms and identifiers used with this specification are described in the separate JSON Web Algorithms (JWA) specification. The JSON Serialization for related encryption functionality is described in the separate JSON Web Encryption JSON Serialization (JWE-JS) specification.

Status of this Memo

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

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

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

This Internet-Draft will expire on April 18, 2013.

Copyright Notice

Copyright (c) 2012 IETF Trust and the persons identified as the

document authors. All rights reserved.

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

Table of Contents

1. Introduction	3
1.1. Notational Conventions	3
2. Terminology	3
3. JSON Serialization	3
4. Example JWS-JS	4
5. IANA Considerations	5
6. Security Considerations	5
7. References	5
7.1. Normative References	5
7.2. Informative References	6
Appendix A. Acknowledgements	6
Appendix B. Open Issues	6
Appendix C. Document History	6
Authors' Addresses	7

1. Introduction

The JSON Web Signature JSON Serialization (JWS-JS) is a format for representing content secured with digital signatures or Message Authentication Codes (MACs) as a JavaScript Object Notation (JSON) [RFC4627] object. It enables multiple digital signatures and/or MACs to be applied to the same content (unlike JWS [JWS]). The digital signature and MAC mechanisms used are independent of the type of content being secured, allowing arbitrary content to be secured. Cryptographic algorithms and identifiers used with this specification are described in the separate JSON Web Algorithms (JWA) [JWA] specification. The JSON Serialization for related encryption functionality is described in the separate JSON Web Encryption JSON Serialization (JWE-JS) [JWE-JS] specification.

1.1. Notational Conventions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in Key words for use in RFCs to Indicate Requirement Levels [RFC2119].

2. Terminology

This specification uses the same terminology as the JSON Web Signature (JWS) [JWS] specification.

3. JSON Serialization

The JSON Serialization represents secured content as a JSON object with a "recipients" member containing an array of per-recipient information and a "payload" member containing a shared Encoded JWS Payload value. Each member of the "recipients" array is a JSON object with a "header" member containing an Encoded JWS Header value and a "signature" member containing an Encoded JWS Signature value.

Unlike the compact serialization used by JWSs, content using the JSON Serialization MAY be secured with more than one digital signature and/or MAC value. Each is represented as an Encoded JWS Signature value in the "signature" member of an object in the "recipients" array. For each, there is an Encoded JWS Encoded Header value in the "header" member of the same object in the "recipients" array. This specifies the digital signature or MAC applied to the Encoded JWS Header value and the shared Encoded JWS Payload value to create the JWS Signature value. Therefore, the syntax is:


```
{ "recipients": [
  { "header": "<header 1 contents>",
    "signature": "<signature 1 contents>" },
  ...
  { "header": "<header N contents>",
    "signature": "<signature N contents>" } ],
  "payload": "<payload contents>"
}
```

The contents of the Encoded JWS Header, Encoded JWS Payload, and Encoded JWS Signature values are exactly as specified in JSON Web Signature (JWS) [JWS]. They are interpreted and validated in the same manner, with each corresponding "header" and "signature" value being created and validated together.

Each JWS Signature value is computed on the JWS Secured Input corresponding to the concatenation of the Encoded JWS Header, a period ('.') character, and the Encoded JWS Payload in the same manner described in the JWS specification. This has the desirable result that each Encoded JWS signature value in the "recipients" array is identical to the value that would be used for the same parameters in a JWS.

4. Example JWS-JS

This section contains an example using the JWS JSON Serialization. This example demonstrates the capability for conveying multiple digital signatures and/or MACs for the same payload.

The Encoded JWS Payload used in this example is the same as used in the examples in Appendix A of JWS (with line breaks for display purposes only):

```
eyJpc3MiOiJqb2UiLA0KICJleHAiOjEzMDA4MTkzODAsDQogImh0dHA6Ly9leGFT
cGx1LmNvbS9pc19yb290Ijp0cnVlfQ
```

Two digital signatures are used in this example: an RSA SHA-256 signature, for which the header and signature values are the same as in Appendix A.2 of JWS, and an ECDSA P-256 SHA-256 signature, for which the header and signature values are the same as in Appendix A.3 of JWS. The two Decoded JWS Header Segments used are:

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

and:

```
{ "alg": "ES256" }
```

Since the computations of the JWS Header and JWS Signature values are the same as in Appendix A.2 and Appendix A.3 of JWS, they are not repeated here.

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

```
{ "recipients": [
  { "header": "eyJhbGciOiJSUzI1NiJ9",
    "signature":
      "cC4hiUPoj9Eetdgtv3hF80EGrhuB__dzERat0XF9g2VtQgr9PJbu3XOiZj5RZ
      mh7AAuHIm4Bh-0Qc_lF5YKt_08W2Fp5jujGbds9uJdbF9CUAr7tldnZcAcQjb
      KBYNX4BAynRFdiuB--f_nZLgrnbyTyWz075vRK5h6xBArLIARNPvkSjtQBMHl
      b1L07Qe7K0GarZRmB_eSN9383LcOLn6_dO--xi12jzDwusC-eOkHWESqtfZES
      c6BfI7noOPqvhJlphCnvWh6IeYI2w9QOYEUpUTI8np6LbgGY9Fs98rqVt5AX
      LIhWkWywlvmtVrBp0igcN_IoypGLUPQGe77Rw" },
  { "header": "eyJhbGciOiJFUzI1NiJ9",
    "signature":
      "DtEhU3ljbEg8L38VWafUAqOyKAM6-Xx-F4GawxaePmXFCgfTjDxw5djxLa8IS
      lSApmWQxfKTUJqPP3-Kg6NU1Q" } ],
  "payload":
    "eyJpc3MiOiJqb2UiLA0KICJleHAiOjEzMDA4MTkzODAsDQogImh0dHA6Ly9leGF
    tcGx1LmNvbS9pc19yb290Ijpo0cnVlQ"
}
```

5. IANA Considerations

This specification makes no requests of IANA.

6. Security Considerations

The security considerations for this specification are the same as those for the JSON Web Signature (JWS) [JWS] specification.

7. References

7.1. Normative References

- [JWA] Jones, M., "JSON Web Algorithms (JWA)", October 2012.
- [JWS] Jones, M., Bradley, J., and N. Sakimura, "JSON Web Signature (JWS)", October 2012.
- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

[RFC4627] Crockford, D., "The application/json Media Type for JavaScript Object Notation (JSON)", RFC 4627, July 2006.

7.2. Informative References

[JSS] Bradley, J. and N. Sakimura (editor), "JSON Simple Sign", September 2010.

[JWE-JS] Jones, M., "JSON Web Encryption JSON Serialization (JWE-JS)", October 2012.

[MagicSignatures] Panzer (editor), J., Laurie, B., and D. Balfanz, "Magic Signatures", January 2011.

Appendix A. Acknowledgements

JSON serializations for secured content were previously explored by Magic Signatures [MagicSignatures] and JSON Simple Sign [JSS].

Appendix B. Open Issues

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

The following items remain to be considered or done in this draft:

- o Track changes that occur in the JWS spec.

Appendix C. Document History

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

-02

- o Changed to use an array of structures for per-recipient values, rather than a set of parallel arrays.

-01

- o Generalized language to refer to Message Authentication Codes (MACs) rather than Hash-based Message Authentication Codes (HMACs).

-00

- o Renamed draft-jones-json-web-signature-json-serialization to draft-jones-jose-jws-json-serialization to have "jose" be in the document name so it can be included in the Related Documents list at <http://datatracker.ietf.org/wg/jose/>. No normative changes.

draft-jones-json-web-signature-json-serialization-02

- o Tracked editorial changes made to the JWS spec.

draft-jones-json-web-signature-json-serialization-01

- o Corrected the Magic Signatures reference.

draft-jones-json-web-signature-json-serialization-00

- o Created the initial version incorporating JOSE working group input and drawing from the JSON Serialization previously proposed in draft-jones-json-web-token-01.

Authors' Addresses

Michael B. Jones
Microsoft

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

John Bradley
independent

Email: ve7jtb@ve7jtb.com

Nat Sakimura
Nomura Research Institute

Email: n-sakimura@nri.co.jp

