

JOSE Working Group
Internet-Draft
Intended status: Standards Track
Expires: April 27, 2015

M. Jones
Microsoft
J. Bradley
Ping Identity
N. Sakimura
NRI
October 24, 2014

JSON Web Signature (JWS)
draft-ietf-jose-json-web-signature-36

Abstract

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

Status of this Memo

This Internet-Draft is submitted in full conformance with the provisions of [BCP 78](#) and [BCP 79](#).

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

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

This Internet-Draft will expire on April 27, 2015.

Copyright Notice

Copyright (c) 2014 IETF Trust and the persons identified as the document authors. All rights reserved.

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

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

Table of Contents

1.	Introduction	5
1.1.	Notational Conventions	5
2.	Terminology	6
3.	JSON Web Signature (JWS) Overview	7
3.1.	JWS Compact Serialization Overview	8
3.2.	JWS JSON Serialization Overview	8
3.3.	Example JWS	9
4.	JOSE Header	10
4.1.	Registered Header Parameter Names	11
4.1.1.	"alg" (Algorithm) Header Parameter	11
4.1.2.	"jku" (JWK Set URL) Header Parameter	11
4.1.3.	"jwk" (JSON Web Key) Header Parameter	11
4.1.4.	"kid" (Key ID) Header Parameter	12
4.1.5.	"x5u" (X.509 URL) Header Parameter	12
4.1.6.	"x5c" (X.509 Certificate Chain) Header Parameter	12
4.1.7.	"x5t" (X.509 Certificate SHA-1 Thumbprint) Header Parameter	13
4.1.8.	"x5t#S256" (X.509 Certificate SHA-256 Thumbprint) Header Parameter	13
4.1.9.	"typ" (Type) Header Parameter	13
4.1.10.	"cty" (Content Type) Header Parameter	14
4.1.11.	"crit" (Critical) Header Parameter	14
4.2.	Public Header Parameter Names	15
4.3.	Private Header Parameter Names	15
5.	Producing and Consuming JWSs	15
5.1.	Message Signature or MAC Computation	15
5.2.	Message Signature or MAC Validation	16
5.3.	String Comparison Rules	18
6.	Key Identification	19
7.	Serializations	19
7.1.	JWS Compact Serialization	20
7.2.	JWS JSON Serialization	20
7.2.1.	General JWS JSON Serialization Syntax	20
7.2.2.	Flattened JWS JSON Serialization Syntax	22
8.	TLS Requirements	23
9.	IANA Considerations	23
9.1.	JSON Web Signature and Encryption Header Parameters Registry	24

9.1.1.	Registration Template	25
9.1.2.	Initial Registry Contents	25
9.2.	Media Type Registration	27
9.2.1.	Registry Contents	27
10.	Security Considerations	28
10.1.	Key Entropy and Random Values	28
10.2.	Key Protection	29
10.3.	Key Origin Authentication	29
10.4.	Cryptographic Agility	29
10.5.	Differences between Digital Signatures and MACs	29
10.6.	Algorithm Validation	30
10.7.	Algorithm Protection	30
10.8.	Chosen Plaintext Attacks	31
10.9.	Timing Attacks	31
10.10.	Replay Protection	31
10.11.	SHA-1 Certificate Thumbprints	31
10.12.	JSON Security Considerations	32
10.13.	Unicode Comparison Security Considerations	32
11.	References	33
11.1.	Normative References	33
11.2.	Informative References	34
Appendix A.	JWS Examples	36
A.1.	Example JWS using HMAC SHA-256	36
A.1.1.	Encoding	36
A.1.2.	Validating	38
A.2.	Example JWS using RSASSA-PKCS-v1_5 SHA-256	38
A.2.1.	Encoding	39
A.2.2.	Validating	41
A.3.	Example JWS using ECDSA P-256 SHA-256	41
A.3.1.	Encoding	41
A.3.2.	Validating	43
A.4.	Example JWS using ECDSA P-521 SHA-512	44
A.4.1.	Encoding	44
A.4.2.	Validating	46
A.5.	Example Unsecured JWS	46
A.6.	Example JWS using General JWS JSON Serialization	47
A.6.1.	JWS Per-Signature Protected Headers	47
A.6.2.	JWS Per-Signature Unprotected Headers	48
A.6.3.	Complete JOSE Header Values	48
A.6.4.	Complete JWS JSON Serialization Representation	48
A.7.	Example JWS using Flattened JWS JSON Serialization	49
Appendix B.	"x5c" (X.509 Certificate Chain) Example	50
Appendix C.	Notes on implementing base64url encoding without padding	51
Appendix D.	Notes on Key Selection	52
Appendix E.	Negative Test Case for "crit" Header Parameter	54
Appendix F.	Detached Content	55
Appendix G.	Acknowledgements	55

Appendix H . Document History	56
Authors' Addresses	66

1. Introduction

JSON Web Signature (JWS) represents content secured with digital signatures or Message Authentication Codes (MACs) using JavaScript Object Notation (JSON) [[RFC7159](#)] based data structures. The JWS cryptographic mechanisms provide integrity protection for an arbitrary sequence of octets. See [Section 10.5](#) for a discussion on the differences between Digital Signatures and MACs.

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

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

Names defined by this specification are short because a core goal is for the resulting representations to be compact.

1.1. Notational Conventions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in Key words for use in RFCs to Indicate Requirement Levels [[RFC2119](#)]. If these words are used without being spelled in uppercase then they are to be interpreted with their normal natural language meanings.

BASE64URL(OCTETS) denotes the base64url encoding of OCTETS, per [Section 2](#).

UTF8(String) denotes the octets of the UTF-8 [[RFC3629](#)] representation of String.

ASCII(String) denotes the octets of the ASCII [[RFC20](#)] representation of String.

The concatenation of two values A and B is denoted as A || B.

2. Terminology

These terms are defined by this specification:

JSON Web Signature (JWS)

A data structure representing a digitally signed or MACed message.

JOSE Header

JSON object containing the parameters describing the cryptographic operations and parameters employed. The JOSE Header is comprised of a set of Header Parameters.

JWS Payload

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

JWS Signature

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

Header Parameter

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

JWS Protected Header

JSON object that contains the Header Parameters that are integrity protected by the JWS Signature digital signature or MAC operation. For the JWS Compact Serialization, this comprises the entire JOSE Header. For the JWS JSON Serialization, this is one component of the JOSE Header.

JWS Unprotected Header

JSON object that contains the Header Parameters that are not integrity protected. This can only be present when using the JWS JSON Serialization.

Base64url Encoding

Base64 encoding using the URL- and filename-safe character set defined in [Section 5 of RFC 4648](#) [RFC4648], with all trailing '=' characters omitted (as permitted by [Section 3.2](#)) and without the inclusion of any line breaks, white space, or other additional characters. Note that the base64url encoding of the empty octet sequence is the empty string. (See [Appendix C](#) for notes on implementing base64url encoding without padding.)

JWS Signing Input

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

JWS Compact Serialization

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

JWS JSON Serialization

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

Unsecured JWS

A JWS object that provides no integrity protection. Unsecured JWSs use the "alg" value "none".

Collision-Resistant Name

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

StringOrURI

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

These terms defined by the JSON Web Encryption (JWE) [[JWE](#)] specification are incorporated into this specification: "JSON Web Encryption (JWE)", "JWE Compact Serialization", and "JWE JSON Serialization".

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

3. JSON Web Signature (JWS) Overview

JWS represents digitally signed or MACed content using JSON data structures and base64url encoding. These JSON data structures MAY contain white space and/or line breaks before or after any JSON values or structural characters, in accordance with [Section 2](#) of RFC

7159 [[RFC7159](#)]. A JWS represents these logical values (each of which is defined in [Section 2](#)):

- o JOSE Header
- o JWS Payload
- o JWS Signature

For a JWS object, the JOSE Header members are the union of the members of these values (each of which is defined in [Section 2](#)):

- o JWS Protected Header
- o JWS Unprotected Header

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

[3.1.](#) JWS Compact Serialization Overview

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

In the JWS Compact Serialization, a JWS object is represented as the concatenation:

```
BASE64URL(UTF8(JWS Protected Header)) || '.' ||  
BASE64URL(JWS Payload) || '.' ||  
BASE64URL(JWS Signature)
```

See [Section 7.1](#) for more information about the JWS Compact Serialization.

[3.2.](#) JWS JSON Serialization Overview

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

In the JWS JSON Serialization, a JWS object is represented as the combination of these four values:


```
BASE64URL(UTF8(JWS Protected Header))
JWS Unprotected Header
BASE64URL(JWS Payload)
BASE64URL(JWS Signature)
```

The three base64url encoded result strings and the JWS Unprotected Header value are represented as members within a JSON object. The inclusion of some of these values is OPTIONAL. The JWS JSON Serialization can also represent multiple signature and/or MAC values, rather than just one. See [Section 7.2](#) for more information about the JWS JSON Serialization.

3.3. Example JWS

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

The following example JWS Protected Header declares that the encoded object is a JSON Web Token (JWT) [[JWT](#)] and the JWS Protected Header and the JWS Payload are secured using the HMAC SHA-256 [[RFC2104](#), [SHS](#)] algorithm:

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

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

```
eyJ0eXAiOiJKV1QiLA0KICJhbGciOiJIUzI1NiJ9
```

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

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

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

```
eyJpc3MiOiJqb2UiLA0KICJleHAiOjEzMDA4MTkzODAsDQogImh0dHA6Ly9leGft
cGxlLmNvbS9pc19yb290Ijp0cnVlfQ
```

Computing the HMAC of the JWS Signing Input ASCII(BASE64URL(UTF8(JWS Protected Header)) || '.' || BASE64URL(JWS Payload)) with the HMAC

SHA-256 algorithm using the key specified in [Appendix A.1](#) and base64url encoding the result yields this BASE64URL(JWS Signature) value:

```
dBjftJeZ4CVP-mB92K27uhbUJU1p1r_wW1gFWFOEjXk
```

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

```
eyJ0eXAiOiJKV1QiLA0KICJhbGciOiJIUzI1NiJ9
.
eyJpc3MiOiJqb2UiLA0KICJleHAiOjEzMDA4MTkzODAsDQogImh0dHA6Ly9leGFT
cGxlLnVhbS9pc19yb290Ijp0cnVlfQ
.
dBjftJeZ4CVP-mB92K27uhbUJU1p1r_wW1gFWFOEjXk
```

See [Appendix A](#) for additional examples, including examples using the JWS JSON Serialization in Sections A.6 and A.7.

4. JOSE Header

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

Implementations are required to understand the specific Header Parameters defined by this specification that are designated as "MUST be understood" and process them in the manner defined in this specification. All other Header Parameters defined by this specification that are not so designated MUST be ignored when not understood. Unless listed as a critical Header Parameter, per [Section 4.1.11](#), all Header Parameters not defined by this specification MUST be ignored when not understood.

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

[4.1.](#) Registered Header Parameter Names

The following Header Parameter names for use in JWS objects are registered in the IANA JSON Web Signature and Encryption Header Parameters registry defined in [Section 9.1](#), with meanings as defined below.

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

[4.1.1.](#) "alg" (Algorithm) Header Parameter

The "alg" (algorithm) Header Parameter identifies the cryptographic algorithm used to secure the JWS. The JWS Signature value is not valid if the "alg" value does not represent a supported algorithm, or if there is not a key for use with that algorithm associated with the party that digitally signed or MACed the content. "alg" values should either be registered in the IANA JSON Web Signature and Encryption Algorithms registry defined in [\[JWA\]](#) or be a value that contains a Collision-Resistant Name. The "alg" value is a case-sensitive string containing a StringOrURI value. This Header Parameter MUST be present and MUST be understood and processed by implementations.

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

[4.1.2.](#) "jku" (JWK Set URL) Header Parameter

The "jku" (JWK Set URL) Header Parameter is a URI [\[RFC3986\]](#) that refers to a resource for a set of JSON-encoded public keys, one of which corresponds to the key used to digitally sign the JWS. The keys MUST be encoded as a JSON Web Key Set (JWK Set) [\[JWK\]](#). The protocol used to acquire the resource MUST provide integrity protection; an HTTP GET request to retrieve the JWK Set MUST use TLS [\[RFC2818, RFC5246\]](#); the identity of the server MUST be validated, as per [Section 6 of RFC 6125](#) [\[RFC6125\]](#). Also, see [Section 8](#) on TLS requirements. Use of this Header Parameter is OPTIONAL.

[4.1.3.](#) "jwk" (JSON Web Key) Header Parameter

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

4.1.4. "kid" (Key ID) Header Parameter

The "kid" (key ID) Header Parameter is a hint indicating which key was used to secure the JWS. This parameter allows originators to explicitly signal a change of key to recipients. The structure of the "kid" value is unspecified. Its value **MUST** be a case-sensitive string. Use of this Header Parameter is **OPTIONAL**.

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

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

The "x5u" (X.509 URL) Header Parameter is a URI [[RFC3986](#)] that refers to a resource for the X.509 public key certificate or certificate chain [[RFC5280](#)] corresponding to the key used to digitally sign the JWS. The identified resource **MUST** provide a representation of the certificate or certificate chain that conforms to [RFC 5280](#) [[RFC5280](#)] in PEM encoded form, with each certificate delimited as specified in [Section 6.1 of RFC 4945](#) [[RFC4945](#)]. The certificate containing the public key corresponding to the key used to digitally sign the JWS **MUST** be the first certificate. This **MAY** be followed by additional certificates, with each subsequent certificate being the one used to certify the previous one. The protocol used to acquire the resource **MUST** provide integrity protection; an HTTP GET request to retrieve the certificate **MUST** use TLS [[RFC2818](#), [RFC5246](#)]; the identity of the server **MUST** be validated, as per [Section 6 of RFC 6125](#) [[RFC6125](#)]. Also, see [Section 8](#) on TLS requirements. Use of this Header Parameter is **OPTIONAL**.

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

The "x5c" (X.509 Certificate Chain) Header Parameter contains the X.509 public key certificate or certificate chain [[RFC5280](#)] corresponding to the key used to digitally sign the JWS. The certificate or certificate chain is represented as a JSON array of certificate value strings. Each string in the array is a base64 encoded ([\[RFC4648\] Section 4](#) -- not base64url encoded) DER [[ITU.X690.1994](#)] PKIX certificate value. The certificate containing the public key corresponding to the key used to digitally sign the JWS **MUST** be the first certificate. This **MAY** be followed by additional certificates, with each subsequent certificate being the one used to certify the previous one. The recipient **MUST** validate the certificate chain according to [RFC 5280](#) [[RFC5280](#)] and reject the signature if any validation failure occurs. Use of this Header Parameter is **OPTIONAL**.

See [Appendix B](#) for an example "x5c" value.

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

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

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

The "x5t#S256" (X.509 Certificate SHA-256 Thumbprint) Header Parameter is a base64url encoded SHA-256 thumbprint (a.k.a. digest) of the DER encoding of the X.509 certificate [[RFC5280](#)] corresponding to the key used to digitally sign the JWS. Note that certificate thumbprints are also sometimes known as certificate fingerprints. Use of this Header Parameter is OPTIONAL.

4.1.9. "typ" (Type) Header Parameter

The "typ" (type) Header Parameter is used by JWS applications to declare the MIME Media Type [[IANA.MediaType](#)] of this complete JWS object. This is intended for use by the application when more than one kind of object could be present in an application data structure that can contain a JWS object; the application can use this value to disambiguate among the different kinds of objects that might be present. It will typically not be used by applications when the kind of object is already known. This parameter is ignored by JWS implementations; any processing of this parameter is performed by the JWS application. Use of this Header Parameter is OPTIONAL.

Per [RFC 2045](#) [[RFC2045](#)], all media type values, subtype values, and parameter names are case-insensitive. However, parameter values are case-sensitive unless otherwise specified for the specific parameter.

To keep messages compact in common situations, it is RECOMMENDED that producers omit an "application/" prefix of a media type value in a "typ" Header Parameter when no other '/' appears in the media type value. A recipient using the media type value MUST treat it as if "application/" were prepended to any "typ" value not containing a '/'. For instance, a "typ" value of "example" SHOULD be used to represent the "application/example" media type; whereas, the media type "application/example;part=1/2" cannot be shortened to "example;part=1/2".

The "typ" value "JOSE" can be used by applications to indicate that this object is a JWS or JWE using the JWS Compact Serialization or

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

4.1.10. "cty" (Content Type) Header Parameter

The "cty" (content type) Header Parameter is used by JWS applications to declare the MIME Media Type [[IANA.MediaType](#)] of the secured content (the payload). This is intended for use by the application when more than one kind of object could be present in the JWS payload; the application can use this value to disambiguate among the different kinds of objects that might be present. It will typically not be used by applications when the kind of object is already known. This parameter is ignored by JWS implementations; any processing of this parameter is performed by the JWS application. Use of this Header Parameter is OPTIONAL.

Per [RFC 2045](#) [[RFC2045](#)], all media type values, subtype values, and parameter names are case-insensitive. However, parameter values are case-sensitive unless otherwise specified for the specific parameter.

To keep messages compact in common situations, it is RECOMMENDED that producers omit an "application/" prefix of a media type value in a "cty" Header Parameter when no other '/' appears in the media type value. A recipient using the media type value MUST treat it as if "application/" were prepended to any "cty" value not containing a '/'. For instance, a "cty" value of "example" SHOULD be used to represent the "application/example" media type; whereas, the media type "application/example;part=1/2" cannot be shortened to "example;part=1/2".

4.1.11. "crit" (Critical) Header Parameter

The "crit" (critical) Header Parameter indicates that extensions to the initial RFC versions of [[this specification]] and [[JWA](#)] are being used that MUST be understood and processed. Its value is an array listing the Header Parameter names present in the JOSE Header that use those extensions. If any of the listed extension Header Parameters are not understood and supported by the recipient, it MUST reject the JWS. Producers MUST NOT include Header Parameter names defined by the initial RFC versions of [[this specification]] or [[JWA](#)] for use with JWS, duplicate names, or names that do not occur as Header Parameter names within the JOSE Header in the "crit" list. Producers MUST NOT use the empty list "[]" as the "crit" value. Recipients MAY reject the JWS if the critical list contains any Header Parameter names defined by the initial RFC versions of [[this specification]] or [[JWA](#)] for use with JWS, or any other constraints

on its use are violated. When used, this Header Parameter MUST be integrity protected; therefore, it MUST occur only within the JWS Protected Header. Use of this Header Parameter is OPTIONAL. This Header Parameter MUST be understood and processed by implementations.

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

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

[4.2.](#) Public Header Parameter Names

Additional Header Parameter names can be defined by those using JWSs. However, in order to prevent collisions, any new Header Parameter name should either be registered in the IANA JSON Web Signature and Encryption Header Parameters registry defined in [Section 9.1](#) or be a Public Name: a value that contains a Collision-Resistant Name. In each case, the definer of the name or value needs to take reasonable precautions to make sure they are in control of the part of the namespace they use to define the Header Parameter name.

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

[4.3.](#) Private Header Parameter Names

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

[5.](#) Producing and Consuming JWSs

[5.1.](#) Message Signature or MAC Computation

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

1. Create the content to be used as the JWS Payload.

2. Compute the encoded payload value `BASE64URL(JWS Payload)`.
3. Create the JSON object(s) containing the desired set of Header Parameters, which together comprise the JOSE Header: if the JWS Compact Serialization is being used, the JWS Protected Header, or if the JWS JSON Serialization is being used, the JWS Protected Header and/or the JWS Unprotected Header.
4. Compute the encoded header value `BASE64URL(UTF8(JWS Protected Header))`. If the JWS Protected Header is not present (which can only happen when using the JWS JSON Serialization and no "protected" member is present), let this value be the empty string.
5. Compute the JWS Signature in the manner defined for the particular algorithm being used over the JWS Signing Input `ASCII(BASE64URL(UTF8(JWS Protected Header)) || '.' || BASE64URL(JWS Payload))`. The "alg" (algorithm) Header Parameter MUST be present in the JOSE Header, with the algorithm value accurately representing the algorithm used to construct the JWS Signature.
6. Compute the encoded signature value `BASE64URL(JWS Signature)`.
7. If the JWS JSON Serialization is being used, repeat this process (steps 3-6) for each digital signature or MAC operation being performed.
8. Create the desired serialized output. The JWS Compact Serialization of this result is `BASE64URL(UTF8(JWS Protected Header)) || '.' || BASE64URL(JWS Payload) || '.' || BASE64URL(JWS Signature)`. The JWS JSON Serialization is described in [Section 7.2](#).

5.2. Message Signature or MAC Validation

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

When there are multiple JWS Signature values, it is an application decision which of the JWS Signature values must successfully validate for the JWS to be accepted. In some cases, all must successfully validate or the JWS will be rejected. In other cases, only a specific JWS signature value needs to be successfully validated. However, in all cases, at least one JWS signature value MUST

successfully validate or the JWS MUST be rejected.

1. Parse the JWS representation to extract the serialized values for the components of the JWS. When using the JWS Compact Serialization, these components are the base64url encoded representations of the JWS Protected Header, the JWS Payload, and the JWS Signature, and when using the JWS JSON Serialization, these components also include the unencoded JWS Unprotected Header value. When using the JWS Compact Serialization, the JWS Protected Header, the JWS Payload, and the JWS Signature are represented as base64url encoded values in that order, with each value being separated from the next by a single period ('.') character, resulting in exactly two delimiting period characters being used. The JWS JSON Serialization is described in [Section 7.2](#).
2. Base64url decode the encoded representation of the JWS Protected Header, following the restriction that no line breaks, white space, or other additional characters have been used.
3. Verify that the resulting octet sequence is a UTF-8 encoded representation of a completely valid JSON object conforming to [RFC 7159](#) [[RFC7159](#)]; let the JWS Protected Header be this JSON object.
4. If using the JWS Compact Serialization, let the JOSE Header be the JWS Protected Header. Otherwise, when using the JWS JSON Serialization, let the JOSE Header be the union of the members of the corresponding JWS Protected Header and JWS Unprotected Header, all of which must be completely valid JSON objects. During this step, verify that the resulting JOSE Header does not contain duplicate Header Parameter names. When using the JWS JSON Serialization, this restriction includes that the same Header Parameter name also MUST NOT occur in distinct JSON object values that together comprise the JOSE Header.
5. Verify that the implementation understands and can process all fields that it is required to support, whether required by this specification, by the algorithm being used, or by the "crit" Header Parameter value, and that the values of those parameters are also understood and supported.
6. Base64url decode the encoded representation of the JWS Payload, following the restriction that no line breaks, white space, or other additional characters have been used.
7. Base64url decode the encoded representation of the JWS Signature, following the restriction that no line breaks, white

space, or other additional characters have been used.

8. Validate the JWS Signature against the JWS Signing Input `ASCII(BASE64URL(UTF8(JWS Protected Header)) || '.' || BASE64URL(JWS Payload))` in the manner defined for the algorithm being used, which **MUST** be accurately represented by the value of the "alg" (algorithm) Header Parameter, which **MUST** be present. See [Section 10.6](#) for security considerations on algorithm validation. Record whether the validation succeeded or not.
9. If the JWS JSON Serialization is being used, repeat this process (steps 4-8) for each digital signature or MAC value contained in the representation.
10. If none of the validations in step 9 succeeded, then the JWS **MUST** be rejected. Otherwise, in the JWS JSON Serialization case, return a result to the application indicating which of the validations succeeded and failed. In the JWS Compact Serialization case, the result can simply indicate whether the JWS was accepted or rejected.

Finally, note that it is an application decision which algorithms may be used in a given context. Even if a JWS can be successfully validated, unless the algorithm(s) used in the JWS are acceptable to the application, it **SHOULD** reject the JWS.

5.3. String Comparison Rules

Processing a JWS inevitably requires comparing known strings to members and values in JSON objects. For example, in checking what the algorithm is, the Unicode string "alg" will be checked against the member names in the JOSE Header to see if there is a matching Header Parameter name. The same process is then used to determine if the value of the "alg" Header Parameter represents a supported algorithm.

The JSON rules for doing member name comparison are described in [Section 8.3 of RFC 7159](#) [RFC7159]. Since the only string comparison operations that are performed are equality and inequality, the same rules can be used for comparing both member names and member values against known strings.

These comparison rules **MUST** be used for all JSON string comparisons except in cases where the definition of the member explicitly calls out that a different comparison rule is to be used for that member value. Only the "typ" and "cty" member values defined in this specification do not use these comparison rules.

Some applications may include case-insensitive information in a case-sensitive value, such as including a DNS name as part of a "kid" (key ID) value. In those cases, the application may need to define a convention for the canonical case to use for representing the case-insensitive portions, such as lowercasing them, if more than one party might need to produce the same value so that they can be compared. (However if all other parties consume whatever value the producing party emitted verbatim without attempting to compare it to an independently produced value, then the case used by the producer will not matter.)

Also, see the JSON security considerations in [Section 10.12](#) and the Unicode security considerations in [Section 10.13](#).

6. Key Identification

It is necessary for the recipient of a JWS to be able to determine the key that was employed for the digital signature or MAC operation. The key employed can be identified using the Header Parameter methods described in [Section 4.1](#) or can be identified using methods that are outside the scope of this specification. Specifically, the Header Parameters "jku", "jwk", "kid", "x5u", "x5c", "x5t", and "x5t#S256" can be used to identify the key used. These Header Parameters MUST be integrity protected if the information that they convey is to be utilized in a trust decision.

The producer SHOULD include sufficient information in the Header Parameters to identify the key used, unless the application uses another means or convention to determine the key used. Validation of the signature or MAC fails when the algorithm used requires a key (which is true of all algorithms except for "none") and the key used cannot be determined.

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

Also, see [Appendix D](#) for notes on possible key selection algorithms.

7. Serializations

JWS objects use one of two serializations, the JWS Compact Serialization or the JWS JSON Serialization. Applications using this specification need to specify what serialization and serialization features are used for that application. For instance, applications might specify that only the JWS JSON Serialization is used, that only JWS JSON Serialization support for a single signature or MAC value is

used, or that support for multiple signatures and/or MAC values is used. JWS implementations only need to implement the features needed for the applications they are designed to support.

7.1. JWS Compact Serialization

The JWS Compact Serialization represents digitally signed or MACed content as a compact, URL-safe string. This string is:

```
BASE64URL(UTF8(JWS Protected Header)) || '.' ||  
BASE64URL(JWS Payload) || '.' ||  
BASE64URL(JWS Signature)
```

Only one signature/MAC is supported by the JWS Compact Serialization and it provides no syntax to represent a JWS Unprotected Header value.

7.2. JWS JSON Serialization

The JWS JSON Serialization represents digitally signed or MACed content as a JSON object. This representation is neither optimized for compactness nor URL-safe.

Two closely related syntaxes are defined for the JWS JSON Serialization: a fully general syntax, with which content can be secured with more than one digital signature and/or MAC operation, and a flattened syntax, which is optimized for the single digital signature or MAC case.

7.2.1. General JWS JSON Serialization Syntax

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

payload

The "payload" member MUST be present and contain the value
BASE64URL(JWS Payload).

signatures

The "signatures" member value MUST be an array of JSON objects.
Each object represents a signature or MAC over the JWS Payload and the JWS Protected Header.

The following members are defined for use in the JSON objects that are elements of the "signatures" array:

protected

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

header

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

signature

The "signature" member MUST be present and contain the value `BASE64URL(JWS Signature)`.

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

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

The Header Parameter values used when creating or validating individual signature or MAC values are the union of the two sets of Header Parameter values that may be present: (1) the JWS Protected Header represented in the "protected" member of the signature/MAC's array element, and (2) the JWS Unprotected Header in the "header" member of the signature/MAC's array element. The union of these sets of Header Parameters comprises the JOSE Header. The Header Parameter names in the two locations MUST be disjoint.

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

In summary, the syntax of a JWS using the general JWS JSON Serialization is as follows:

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

See [Appendix A.6](#) for an example JWS using the general JWS JSON Serialization syntax.

7.2.2. Flattened JWS JSON Serialization Syntax

The flattened JWS JSON Serialization syntax is based upon the general syntax, but flattens it, optimizing it for the single digital signature/MAC case. It flattens it by removing the "signatures" member and instead placing those members defined for use in the "signatures" array (the "protected", "header", and "signature" members) in the top-level JSON object (at the same level as the "payload" member).

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

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

```
{
  "payload":"<payload contents>",
  "protected":"<integrity-protected header contents>",
  "header":"<non-integrity-protected header contents>",
  "signature":"<signature contents>"
}
```

See [Appendix A.7](#) for an example JWS using the flattened JWS JSON Serialization syntax.

8. TLS Requirements

Implementations MUST support TLS. Which version(s) ought to be implemented will vary over time, and depend on the widespread deployment and known security vulnerabilities at the time of implementation. At the time of this writing, TLS version 1.2 [[RFC5246](#)] is the most recent version.

To protect against information disclosure and tampering, confidentiality protection MUST be applied using TLS with a ciphersuite that provides confidentiality and integrity protection. See current publications by the IETF TLS working group, including [RFC 6176](#) [[RFC6176](#)], for guidance on the ciphersuites currently considered to be appropriate for use. Also, see Recommendations for Secure Use of TLS and DTLS [[I-D.ietf-uta-tls-bcp](#)] for recommendations on improving the security of software and services using TLS.

Whenever TLS is used, the identity of the service provider encoded in the TLS server certificate MUST be verified using the procedures described in [Section 6 of RFC 6125](#) [[RFC6125](#)]. TLS is used by the "jku" and "x5u" Header Parameters defined by this specification.

9. IANA Considerations

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

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

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

Within the review period, the Designated Expert(s) will either approve or deny the registration request, communicating this decision to the review list and IANA. Denials should include an explanation and, if applicable, suggestions as to how to make the request successful. Registration requests that are undetermined for a period longer than 21 days can be brought to the IESG's attention (using the iesg@ietf.org mailing list) for resolution.

Criteria that should be applied by the Designated Expert(s) includes

determining whether the proposed registration duplicates existing functionality, determining whether it is likely to be of general applicability or whether it is useful only for a single application, and whether the registration description is clear.

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

It is suggested that multiple Designated Experts be appointed who are able to represent the perspectives of different applications using this specification, in order to enable broadly-informed review of registration decisions. In cases where a registration decision could be perceived as creating a conflict of interest for a particular Expert, that Expert should defer to the judgment of the other Expert(s).

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

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

]]

9.1. JSON Web Signature and Encryption Header Parameters Registry

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

9.1.1. Registration Template

Header Parameter Name:

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

Header Parameter Description:

Brief description of the Header Parameter (e.g., "Key ID").

Header Parameter Usage Location(s):

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

Change Controller:

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

Specification Document(s):

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

9.1.2. Initial Registry Contents

This specification registers the Header Parameter names defined in [Section 4.1](#) in this registry.

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

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

- o Header Parameter Name: "jwk"
- o Header Parameter Description: JSON Web Key
- o Header Parameter Usage Location(s): JWS
- o Change Controller: IESG
- o Specification document(s): [Section 4.1.3](#) of [[this document]]
- o Header Parameter Name: "kid"
- o Header Parameter Description: Key ID
- o Header Parameter Usage Location(s): JWS
- o Change Controller: IESG
- o Specification Document(s): [Section 4.1.4](#) of [[this document]]
- o Header Parameter Name: "x5u"
- o Header Parameter Description: X.509 URL
- o Header Parameter Usage Location(s): JWS
- o Change Controller: IESG
- o Specification Document(s): [Section 4.1.5](#) of [[this document]]
- o Header Parameter Name: "x5c"
- o Header Parameter Description: X.509 Certificate Chain
- o Header Parameter Usage Location(s): JWS
- o Change Controller: IESG
- o Specification Document(s): [Section 4.1.6](#) of [[this document]]
- o Header Parameter Name: "x5t"
- o Header Parameter Description: X.509 Certificate SHA-1 Thumbprint
- o Header Parameter Usage Location(s): JWS
- o Change Controller: IESG
- o Specification Document(s): [Section 4.1.7](#) of [[this document]]
- o Header Parameter Name: "x5t#S256"
- o Header Parameter Description: X.509 Certificate SHA-256 Thumbprint
- o Header Parameter Usage Location(s): JWS
- o Change Controller: IESG
- o Specification Document(s): [Section 4.1.8](#) of [[this document]]
- o Header Parameter Name: "typ"
- o Header Parameter Description: Type
- o Header Parameter Usage Location(s): JWS
- o Change Controller: IESG
- o Specification Document(s): [Section 4.1.9](#) of [[this document]]
- o Header Parameter Name: "cty"
- o Header Parameter Description: Content Type
- o Header Parameter Usage Location(s): JWS
- o Change Controller: IESG

- o Specification Document(s): [Section 4.1.10](#) of [[this document]]
- o Header Parameter Name: "crit"
- o Header Parameter Description: Critical
- o Header Parameter Usage Location(s): JWS
- o Change Controller: IESG
- o Specification Document(s): [Section 4.1.11](#) of [[this document]]

[9.2.](#) Media Type Registration

[9.2.1.](#) Registry Contents

This specification registers the "application/jose" Media Type [[RFC2046](#)] in the MIME Media Types registry [[IANA.MediaTypees](#)] in the manner described in [RFC 6838](#) [[RFC6838](#)], which can be used to indicate that the content is a JWS or JWE object using the JWS Compact Serialization or the JWE Compact Serialization and the "application/jose+json" Media Type in the MIME Media Types registry, which can be used to indicate that the content is a JWS or JWE object using the JWS JSON Serialization or the JWE JSON Serialization.

- o Type name: application
- o Subtype name: jose
- o Required parameters: n/a
- o Optional parameters: n/a
- o Encoding considerations: 8bit; application/jose values are encoded as a series of base64url encoded values (some of which may be the empty string) each separated from the next by a single period ('.') character.
- o Security considerations: See the Security Considerations section of [[this document]]
- o Interoperability considerations: n/a
- o Published specification: [[this document]]
- o Applications that use this media type: OpenID Connect, Mozilla Persona, Salesforce, Google, Android, Windows Azure, Xbox One, and numerous others that use JWTs
- o Fragment identifier considerations: n/a
- o Additional information: Magic number(s): n/a, File extension(s): n/a, Macintosh file type code(s): n/a
- o Person & email address to contact for further information: Michael B. Jones, mbj@microsoft.com
- o Intended usage: COMMON
- o Restrictions on usage: none
- o Author: Michael B. Jones, mbj@microsoft.com
- o Change Controller: IESG
- o Provisional registration? No

- o Type name: application
- o Subtype name: jose+json
- o Required parameters: n/a
- o Optional parameters: n/a
- o Encoding considerations: 8bit; application/jose+json values are represented as a JSON Object; UTF-8 encoding SHOULD be employed for the JSON object.
- o Security considerations: See the Security Considerations section of [[this document]]
- o Interoperability considerations: n/a
- o Published specification: [[this document]]
- o Applications that use this media type: TBD
- o Fragment identifier considerations: n/a
- o Additional information: Magic number(s): n/a, File extension(s): n/a, Macintosh file type code(s): n/a
- o Person & email address to contact for further information: Michael B. Jones, mbj@microsoft.com
- o Intended usage: COMMON
- o Restrictions on usage: none
- o Author: Michael B. Jones, mbj@microsoft.com
- o Change Controller: IESG
- o Provisional registration? No

10. Security Considerations

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

All the security considerations in XML DSIG 2.0 [\[W3C.NOTE-xmlsig-core2-20130411\]](#), also apply to this specification, other than those that are XML specific. Likewise, many of the best practices documented in XML Signature Best Practices [\[W3C.NOTE-xmlsig-bestpractices-20130411\]](#) also apply to this specification, other than those that are XML specific.

10.1. Key Entropy and Random Values

Keys are only as strong as the amount of entropy used to generate them. A minimum of 128 bits of entropy should be used for all keys, and depending upon the application context, more may be required.

Implementations must randomly generate public/private key pairs, message authentication (MAC) keys, and padding values. The use of inadequate pseudo-random number generators (PRNGs) to generate cryptographic keys can result in little or no security. An attacker

may find it much easier to reproduce the PRNG environment that produced the keys, searching the resulting small set of possibilities, rather than brute force searching the whole key space. The generation of quality random numbers is difficult. [RFC 4086](#) [RFC4086] offers important guidance in this area.

[10.2.](#) Key Protection

Implementations must protect the signer's private key. Compromise of the signer's private key permits an attacker to masquerade as the signer.

Implementations must protect the message authentication (MAC) key. Compromise of the MAC key may result in undetectable modification of the authenticated content.

[10.3.](#) Key Origin Authentication

The key management technique employed to obtain public keys must authenticate the origin of the key; otherwise, it is unknown what party signed the message.

Likewise, the key management technique employed to distribute MAC keys must provide data origin authentication; otherwise, the contents are delivered with integrity from an unknown source.

[10.4.](#) Cryptographic Agility

See Section 8.1 of [\[JWA\]](#) for security considerations on cryptographic agility.

[10.5.](#) Differences between Digital Signatures and MACs

While MACs and digital signatures can both be used for integrity checking, there are some significant differences between the security properties that each of them provides. These need to be taken into consideration when designing protocols and selecting the algorithms to be used in protocols.

Both signatures and MACs provide for integrity checking -- verifying that the message has not been modified since the integrity value was computed. However, MACs provide for origination identification only under specific circumstances. It can normally be assumed that a private key used for a signature is only in the hands of a single entity (although perhaps a distributed entity, in the case of replicated servers); however, a MAC key needs to be in the hands of all the entities that use it for integrity computation and checking. Validation of a MAC only provides corroboration that the message was

generated by one of the parties that knows the symmetric MAC key. This means that origination can only be determined if a MAC key is known only to two entities and the recipient knows that it did not create the message. MAC validation cannot be used to prove origination to a third party.

10.6. Algorithm Validation

The digital signature representations for some algorithms include information about the algorithm used inside the signature value. For instance, signatures produced with RSASSA-PKCS-v1_5 [[RFC3447](#)] encode the hash function used and many libraries actually use the hash algorithm specified inside the signature when validating the signature. When using such libraries, as part of the algorithm validation performed, implementations MUST ensure that the algorithm information encoded in the signature corresponds to that specified with the "alg" Header Parameter. If this is not done, an attacker could claim to have used a strong hash algorithm while actually using a weak one represented in the signature value.

10.7. Algorithm Protection

In some usages of JWS, there is a risk of algorithm substitution attacks, in which an attacker can use an existing digital signature value with a different signature algorithm to make it appear that a signer has signed something that it has not. These attacks have been discussed in detail in the context of CMS [[RFC6211](#)]. This risk arises when all of the following are true:

- o Verifiers of a signature support multiple algorithms.
- o Given an existing signature, an attacker can find another payload that produces the same signature value with a different algorithm.
- o The payload crafted by the attacker is valid in the application context.

There are several ways for an application to mitigate algorithm substitution attacks:

- o Use only digital signature algorithms that are not vulnerable to substitution attacks. Substitution attacks are only feasible if an attacker can compute pre-images for a hash function accepted by the recipient. All JWA-defined signature algorithms use SHA-2 hashes, for which there are no known pre-image attacks, as of the time of this writing.

- o Require that the "alg" Header Parameter be carried in the protected header. (This is always the case when using the JWS Compact Serialization and is the approach taken by CMS [[RFC6211](#)].)
- o Include a field containing the algorithm in the application payload, and require that it be matched with the "alg" Header Parameter during verification. (This is the approach taken by PKIX [[RFC5280](#)].)

[10.8.](#) Chosen Plaintext Attacks

Creators of JWSs should not allow third parties to insert arbitrary content into the message without adding entropy not controlled by the third party.

[10.9.](#) Timing Attacks

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

[10.10.](#) Replay Protection

While not directly in scope for this specification, note that applications using JWS (or JWE) objects can thwart replay attacks by including a unique message identifier as integrity protected content in the JWS (or JWE) message and having the recipient verify that the message has not been previously received or acted upon.

[10.11.](#) SHA-1 Certificate Thumbprints

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

Alternatively, the "x5t#S256" (X.509 Certificate SHA-256 Thumbprint) Header Parameter could be used instead of "x5t". However, at the time of this writing, no development platform is known to support SHA-256 certificate thumbprints.

10.12. JSON Security Considerations

Strict JSON [[RFC7159](#)] validation is a security requirement. If malformed JSON is received, then the intent of the producer is impossible to reliably discern. Ambiguous and potentially exploitable situations could arise if the JSON parser used does not reject malformed JSON syntax. In particular, any JSON inputs not conforming to the JSON-text syntax defined in [RFC 7159](#) input MUST be rejected in their entirety.

[Section 4](#) of the JSON Data Interchange Format specification [[RFC7159](#)] states "The names within an object SHOULD be unique", whereas this specification states that "Header Parameter names within this object MUST be unique; JWS parsers MUST either reject JWSs with duplicate Header Parameter names or use a JSON parser that returns only the lexically last duplicate member name, as specified in [Section 15.12](#) (The JSON Object) of ECMA Script 5.1 [[ECMAScript](#)]". Thus, this specification requires that the [Section 4](#) "SHOULD" be treated as a "MUST" by producers and that it be either treated as a "MUST" or in the manner specified in ECMA Script 5.1 by consumers. Ambiguous and potentially exploitable situations could arise if the JSON parser used does not enforce the uniqueness of member names or returns an unpredictable value for duplicate member names.

Some JSON parsers might not reject input that contains extra significant characters after a valid input. For instance, the input `{"tag":"value"}ABCD` contains a valid JSON-text object followed by the extra characters "ABCD". Such input MUST be rejected in its entirety.

10.13. Unicode Comparison Security Considerations

Header Parameter names and algorithm names are Unicode strings. For security reasons, the representations of these names must be compared verbatim after performing any escape processing (as per [Section 8.3 of RFC 7159](#) [[RFC7159](#)]). This means, for instance, that these JSON strings must compare as being equal (`"sig"`, `"\u0073ig"`), whereas these must all compare as being not equal to the first set or to each other (`"SIG"`, `"Sig"`, `"si\u0047"`).

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

11. References

11.1. Normative References

[ECMAScript]

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

[IANA.MediaType]

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

[ITU.X690.1994]

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

[JWA]

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

[JWK]

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

[RFC20]

Cerf, V., "ASCII format for Network Interchange", [RFC 20](#), October 1969.

[RFC2045]

Freed, N. and N. Borenstein, "Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies", [RFC 2045](#), November 1996.

[RFC2046]

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

[RFC2119]

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

[RFC2818]

Rescorla, E., "HTTP Over TLS", [RFC 2818](#), May 2000.

[RFC3629]

Yergeau, F., "UTF-8, a transformation format of ISO 10646", STD 63, [RFC 3629](#), November 2003.

[RFC3986]

Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax", STD 66,

[RFC 3986](#), January 2005.

- [RFC4648] Josefsson, S., "The Base16, Base32, and Base64 Data Encodings", [RFC 4648](#), October 2006.
- [RFC4945] Korver, B., "The Internet IP Security PKI Profile of IKEv1/ISAKMP, IKEv2, and PKIX", [RFC 4945](#), August 2007.
- [RFC4949] Shirey, R., "Internet Security Glossary, Version 2", [RFC 4949](#), August 2007.
- [RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", [RFC 5246](#), August 2008.
- [RFC5280] Cooper, D., Santesson, S., Farrell, S., Boeyen, S., Housley, R., and W. Polk, "Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile", [RFC 5280](#), May 2008.
- [RFC6125] Saint-Andre, P. and J. Hodges, "Representation and Verification of Domain-Based Application Service Identity within Internet Public Key Infrastructure Using X.509 (PKIX) Certificates in the Context of Transport Layer Security (TLS)", [RFC 6125](#), March 2011.
- [RFC6176] Turner, S. and T. Polk, "Prohibiting Secure Sockets Layer (SSL) Version 2.0", [RFC 6176](#), March 2011.
- [RFC7159] Bray, T., "The JavaScript Object Notation (JSON) Data Interchange Format", [RFC 7159](#), March 2014.

[11.2.](#) Informative References

- [CanvasApp] Facebook, "Canvas Applications", 2010.
- [I-D.ietf-uta-tls-bcp] Sheffer, Y., Holz, R., and P. Saint-Andre, "Recommendations for Secure Use of TLS and DTLS", [draft-ietf-uta-tls-bcp-06](#) (work in progress), October 2014.
- [JSS] Bradley, J. and N. Sakimura (editor), "JSON Simple Sign", September 2010.
- [JWE] Jones, M. and J. Hildebrand, "JSON Web Encryption (JWE)", [draft-ietf-jose-json-web-encryption](#) (work in progress), October 2014.

- [JWT] Jones, M., Bradley, J., and N. Sakimura, "JSON Web Token (JWT)", [draft-ietf-oauth-json-web-token](#) (work in progress), October 2014.
- [MagicSignatures]
Panzer (editor), J., Laurie, B., and D. Balfanz, "Magic Signatures", January 2011.
- [RFC2104] Krawczyk, H., Bellare, M., and R. Canetti, "HMAC: Keyed-Hashing for Message Authentication", [RFC 2104](#), February 1997.
- [RFC3447] Jonsson, J. and B. Kaliski, "Public-Key Cryptography Standards (PKCS) #1: RSA Cryptography Specifications Version 2.1", [RFC 3447](#), February 2003.
- [RFC4086] Eastlake, D., Schiller, J., and S. Crocker, "Randomness Requirements for Security", [BCP 106](#), [RFC 4086](#), June 2005.
- [RFC4122] Leach, P., Mealling, M., and R. Salz, "A Universally Unique IDentifier (UUID) URN Namespace", [RFC 4122](#), July 2005.
- [RFC5226] Narten, T. and H. Alvestrand, "Guidelines for Writing an IANA Considerations Section in RFCs", [BCP 26](#), [RFC 5226](#), May 2008.
- [RFC6211] Schaad, J., "Cryptographic Message Syntax (CMS) Algorithm Identifier Protection Attribute", [RFC 6211](#), April 2011.
- [RFC6838] Freed, N., Klensin, J., and T. Hansen, "Media Type Specifications and Registration Procedures", [BCP 13](#), [RFC 6838](#), January 2013.
- [SHS] National Institute of Standards and Technology, "Secure Hash Standard (SHS)", FIPS PUB 180-4, March 2012.
- [W3C.NOTE-xmlldsig-bestpractices-20130411]
Hirsch, F. and P. Datta, "XML Signature Best Practices", World Wide Web Consortium Note NOTE-xmlldsig-bestpractices-20130411, April 2013, <<http://www.w3.org/TR/2013/NOTE-xmlldsig-bestpractices-20130411/>>.
- [W3C.NOTE-xmlldsig-core2-20130411]
Eastlake, D., Reagle, J., Solo, D., Hirsch, F., Roessler, T., Yiu, K., Datta, P., and S. Cantor, "XML Signature Syntax and Processing Version 2.0", World Wide Web Consortium Note NOTE-xmlldsig-core2-20130411, April 2013,

<<http://www.w3.org/TR/2013/NOTE-xmlsig-core2-20130411/>>.

[Appendix A.](#) JWS Examples

This section provides several examples of JWSs. While the first three examples all represent JSON Web Tokens (JWTs) [[JWT](#)], the payload can be any octet sequence, as shown in [Appendix A.4](#).

[A.1.](#) Example JWS using HMAC SHA-256

[A.1.1.](#) Encoding

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

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

To remove potential ambiguities in the representation of the JSON object above, the actual octet sequence representing UTF8(JWS Protected Header) used in this example is also included below. (Note that ambiguities can arise due to differing platform representations of line breaks (CRLF versus LF), differing spacing at the beginning and ends of lines, whether the last line has a terminating line break or not, and other causes. In the representation used in this example, the first line has no leading or trailing spaces, a CRLF line break (13, 10) occurs between the first and second lines, the second line has one leading space (32) and no trailing spaces, and the last line does not have a terminating line break.) The octets representing UTF8(JWS Protected Header) in this example (using JSON array notation) are:

```
[123, 34, 116, 121, 112, 34, 58, 34, 74, 87, 84, 34, 44, 13, 10, 32,  
34, 97, 108, 103, 34, 58, 34, 72, 83, 50, 53, 54, 34, 125]
```

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

```
eyJ0eXAiOiJKV1QiLA0KICJhbGciOiJIUzI1NiJ9
```

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


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

The following octet sequence, which is the UTF-8 representation used in this example for the JSON object above, is the JWS Payload:

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

Encoding this JWS Protected Header as `BASE64URL(UTF8(JWS Protected Header))` gives this value (with line breaks for display purposes only):

```
eyJpc3MiOiJqb2UiLA0KICJleHAiOjEzMDA4MTkzODAsDQogImh0dHA6Ly9leGft
cGx1LmNvbS9pc19yb290Ijp0cnVlfQ
```

Combining these as `BASE64URL(UTF8(JWS Protected Header)) || '.' || BASE64URL(JWS Payload)` gives this string (with line breaks for display purposes only):

```
eyJ0eXAiOiJKV1QiLA0KICJhbGciOiJIUzI1NiJ9
.
eyJpc3MiOiJqb2UiLA0KICJleHAiOjEzMDA4MTkzODAsDQogImh0dHA6Ly9leGft
cGx1LmNvbS9pc19yb290Ijp0cnVlfQ
```

The resulting JWS Signing Input value, which is the ASCII representation of above string, is the following octet sequence (using JSON array notation):

```
[101, 121, 74, 48, 101, 88, 65, 105, 79, 105, 74, 75, 86, 49, 81,
105, 76, 65, 48, 75, 73, 67, 74, 104, 98, 71, 99, 105, 79, 105, 74,
73, 85, 122, 73, 49, 78, 105, 74, 57, 46, 101, 121, 74, 112, 99, 51,
77, 105, 79, 105, 74, 113, 98, 50, 85, 105, 76, 65, 48, 75, 73, 67,
74, 108, 101, 72, 65, 105, 79, 106, 69, 122, 77, 68, 65, 52, 77, 84,
107, 122, 79, 68, 65, 115, 68, 81, 111, 103, 73, 109, 104, 48, 100,
72, 65, 54, 76, 121, 57, 108, 101, 71, 70, 116, 99, 71, 120, 108, 76,
109, 78, 118, 98, 83, 57, 112, 99, 49, 57, 121, 98, 50, 57, 48, 73,
106, 112, 48, 99, 110, 86, 108, 102, 81]
```

HMACs are generated using keys. This example uses the symmetric key represented in JSON Web Key [[JWK](#)] format below (with line breaks within values for display purposes only):


```
{ "kty": "oct",  
  "k": "AyM1SysPpbyDfgZld3umj1qzK0bwVMkoqQ-EstJQLr_T-1qS0gZH75  
      aKtMN3Yj0iPS4hcgUuTwjAzZr1Z9CAow"  
}
```

Running the HMAC SHA-256 algorithm on the JWS Signing Input with this key yields this JWS Signature octet sequence:

```
[116, 24, 223, 180, 151, 153, 224, 37, 79, 250, 96, 125, 216, 173,  
187, 186, 22, 212, 37, 77, 105, 214, 191, 240, 91, 88, 5, 88, 83,  
132, 141, 121]
```

Encoding this JWS Signature as BASE64URL(JWS Signature) gives this value:

```
dBjftJeZ4CVP-mB92K27uhbUJU1p1r_wW1gFWF0EjXk
```

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

```
eyJ0eXAiOiJKV1QiLA0KICJhbGciOiJIUzI1NiJ9  
.  
eyJpc3MiOiJqb2UiLA0KICJleHAiOjEzMDA4MTkzODAsDQogImh0dHA6Ly9leGFt  
cGx1LmNvbS9pc19yb290Ijp0cnVlfQ  
.  
dBjftJeZ4CVP-mB92K27uhbUJU1p1r_wW1gFWF0EjXk
```

[A.1.2.](#) Validating

Since the "alg" Header Parameter is "HS256", we validate the HMAC SHA-256 value contained in the JWS Signature.

To validate the HMAC value, we repeat the previous process of using the correct key and the JWS Signing Input (which is the initial substring of the JWS Compact Serialization representation up until but not including the second period character) as input to the HMAC SHA-256 function and then taking the output and determining if it matches the JWS Signature (which is base64url decoded from the value encoded in the JWS representation). If it matches exactly, the HMAC has been validated.

[A.2.](#) Example JWS using RSASSA-PKCS-v1_5 SHA-256

[A.2.1.](#) Encoding

The JWS Protected Header in this example is different from the previous example in two ways: First, because a different algorithm is being used, the "alg" value is different. Second, for illustration purposes only, the optional "typ" parameter is not used. (This difference is not related to the algorithm employed.) The JWS Protected Header used is:

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

The octets representing UTF8(JWS Protected Header) in this example (using JSON array notation) are:

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

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

```
eyJhbGciOiJSUzI1NiJ9
```

The JWS Payload used in this example, which follows, is the same as in the previous example. Since the BASE64URL(JWS Payload) value will therefore be the same, its computation is not repeated here.

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

Combining these as BASE64URL(UTF8(JWS Protected Header)) || '.' || BASE64URL(JWS Payload) gives this string (with line breaks for display purposes only):

```
eyJhbGciOiJSUzI1NiJ9  
.  
eyJpc3MiOiJqb2UiLA0KICJleHAiOjEzMDA4MTkzODAsDQogImh0dHA6Ly9leGFt  
cGx1LnNvbS9pc19yb290Ijp0cnVlfQ
```

The resulting JWS Signing Input value, which is the ASCII representation of above string, is the following octet sequence:

```
[101, 121, 74, 104, 98, 71, 99, 105, 79, 105, 74, 83, 85, 122, 73,  
49, 78, 105, 74, 57, 46, 101, 121, 74, 112, 99, 51, 77, 105, 79, 105,  
74, 113, 98, 50, 85, 105, 76, 65, 48, 75, 73, 67, 74, 108, 101, 72,  
65, 105, 79, 106, 69, 122, 77, 68, 65, 52, 77, 84, 107, 122, 79, 68,  
65, 115, 68, 81, 111, 103, 73, 109, 104, 48, 100, 72, 65, 54, 76,  
121, 57, 108, 101, 71, 70, 116, 99, 71, 120, 108, 76, 109, 78, 118,  
98, 83, 57, 112, 99, 49, 57, 121, 98, 50, 57, 48, 73, 106, 112, 48,
```


99, 110, 86, 108, 102, 81]

This example uses the RSA key represented in JSON Web Key [JWK] format below (with line breaks within values for display purposes only):

```
{
  "kty": "RSA",
  "n": "ofgWCuLjybRlzo0tZWJjNiuSfb4p4fAkd_wWJcyQoTbj9k0l8W26mPddx
HmfHQp-Vaw-4qPCJrcS2mJPMEzP1Pt0Bm4d4Q1L-yRT-SFd2lZS-pCgNMs
D1W_YpRPEwOWvG6b32690r2jZ47soMZo9wGzjb_70Mg0LOL-bSf63kpaSH
SXndS5z5rexMdbBYUsLA9e-KXBdQOS-UTo7WTBEMa2R2CapHg665xsmtDV
MTBQY4uDZ1xvb3qCo5ZWKh9kG4LT6_I5Ih1JH7aGhyxXFvUK-DWNmoudF8
NAco9_h9iaGNj8q2ethFkMLs91kzk2PAcDTW9gb54h4FRWyuXpoQ",
  "e": "AQAB",
  "d": "Eq5xpGnNCivDf1JsRQBxHx1hdR1k6U1we2JZD50LpXyWPEAeP88vLN097I
j1A7_GQ5sLKMgvfTeXZx9SE-7YwVol2NX0oAJe46sui395IW_GO-pWJ100
BktGoVEn2bKVRUCgu-GjBVaYLU6f3l9kJfFNS3E0QbVdxzubSu3Mkqzjkn
439X0M_V51gfPRLI9JYanrC4D4qAdGcopV_0ZHHzQlBjudU2QvXt4ehNYT
CBR6XCLQUShb1juU01ZdiYoFaFQT5Tw8bGU1_x_jTj3ccPDVZFD9pIuhLh
B0neufuBiB4cS98l2SR_RQyGWSewjnczT0QU91p1Dh0VRuOopznQ"
}
```

The RSA private key is then passed to the RSA signing function, which also takes the hash type, SHA-256, and the JWS Signing Input as inputs. The result of the digital signature is an octet sequence, which represents a big endian integer. In this example, it is:

```
[112, 46, 33, 137, 67, 232, 143, 209, 30, 181, 216, 45, 191, 120, 69,
243, 65, 6, 174, 27, 129, 255, 247, 115, 17, 22, 173, 209, 113, 125,
131, 101, 109, 66, 10, 253, 60, 150, 238, 221, 115, 162, 102, 62, 81,
102, 104, 123, 0, 11, 135, 34, 110, 1, 135, 237, 16, 115, 249, 69,
229, 130, 173, 252, 239, 22, 216, 90, 121, 142, 232, 198, 109, 219,
61, 184, 151, 91, 23, 208, 148, 2, 190, 237, 213, 217, 217, 112, 7,
16, 141, 178, 129, 96, 213, 248, 4, 12, 167, 68, 87, 98, 184, 31,
190, 127, 249, 217, 46, 10, 231, 111, 36, 242, 91, 51, 187, 230, 244,
74, 230, 30, 177, 4, 10, 203, 32, 4, 77, 62, 249, 18, 142, 212, 1,
48, 121, 91, 212, 189, 59, 65, 238, 202, 208, 102, 171, 101, 25, 129,
253, 228, 141, 247, 127, 55, 45, 195, 139, 159, 175, 221, 59, 239,
177, 139, 93, 163, 204, 60, 46, 176, 47, 158, 58, 65, 214, 18, 202,
173, 21, 145, 18, 115, 160, 95, 35, 185, 232, 56, 250, 175, 132, 157,
105, 132, 41, 239, 90, 30, 136, 121, 130, 54, 195, 212, 14, 96, 69,
34, 165, 68, 200, 242, 122, 122, 45, 184, 6, 99, 209, 108, 247, 202,
234, 86, 222, 64, 92, 178, 33, 90, 69, 178, 194, 85, 102, 181, 90,
193, 167, 72, 160, 112, 223, 200, 163, 42, 70, 149, 67, 208, 25, 238,
251, 71]
```

Encoding the signature as BASE64URL(JWS Signature) produces this value (with line breaks for display purposes only):


```
cC4hiUPoj9Eetdgtv3hF80EGrhuB__dzERat0XF9g2VtQgr9PJbu3X0iZj5RZmh7
AAuHIm4Bh-0Qc_lF5YKt_08W2Fp5jujGbdS9uJdbF9CUAr7t1dnZcAcQjbKBYNX4
BAynRFdiuB--f_nZLgrnbyTyWz075vRK5h6xBarLIARNPvkSjtQBMHlb1L07Qe7K
0GarZRmB_eSN9383Lc0Ln6_d0--xi12jzDwusC-e0kHWEsqtfZESc6BfI7no0Pqv
hJ1phCnvWh6IeYI2w9Q0YEUipUTI8np6LbgGY9Fs98rqVt5AXLIhWkWyw1VmtVrB
p0igcN_IoypGlUPQGe77Rw
```

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

```
eyJhbGciOiJSUzI1NiJ9
.
eyJpc3MiOiJqb2UiLA0KICJleHAiOjEzMDA4MTkzODAsDQogImh0dHA6Ly9leGFt
cGx1LmNvbS9pc19yb290Ijp0cnVlfQ
.
cC4hiUPoj9Eetdgtv3hF80EGrhuB__dzERat0XF9g2VtQgr9PJbu3X0iZj5RZmh7
AAuHIm4Bh-0Qc_lF5YKt_08W2Fp5jujGbdS9uJdbF9CUAr7t1dnZcAcQjbKBYNX4
BAynRFdiuB--f_nZLgrnbyTyWz075vRK5h6xBarLIARNPvkSjtQBMHlb1L07Qe7K
0GarZRmB_eSN9383Lc0Ln6_d0--xi12jzDwusC-e0kHWEsqtfZESc6BfI7no0Pqv
hJ1phCnvWh6IeYI2w9Q0YEUipUTI8np6LbgGY9Fs98rqVt5AXLIhWkWyw1VmtVrB
p0igcN_IoypGlUPQGe77Rw
```

[A.2.2.](#) Validating

Since the "alg" Header Parameter is "RS256", we validate the RSASSA-PKCS-v1_5 SHA-256 digital signature contained in the JWS Signature.

Validating the JWS Signature is a bit different from the previous example. We pass the public key (n, e), the JWS Signature (which is base64url decoded from the value encoded in the JWS representation), and the JWS Signing Input (which is the initial substring of the JWS Compact Serialization representation up until but not including the second period character) to an RSASSA-PKCS-v1_5 signature verifier that has been configured to use the SHA-256 hash function.

[A.3.](#) Example JWS using ECDSA P-256 SHA-256

[A.3.1.](#) Encoding

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

```
{"alg":"ES256"}
```

The octets representing UTF8(JWS Protected Header) in this example

(using JSON array notation) are:

```
[123, 34, 97, 108, 103, 34, 58, 34, 69, 83, 50, 53, 54, 34, 125]
```

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

```
eyJhbGciOiJFUzI1NiJ9
```

The JWS Payload used in this example, which follows, is the same as in the previous examples. Since the `BASE64URL(JWS Payload)` value will therefore be the same, its computation is not repeated here.

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

Combining these as `BASE64URL(UTF8(JWS Protected Header)) || '.' || BASE64URL(JWS Payload)` gives this string (with line breaks for display purposes only):

```
eyJhbGciOiJFUzI1NiJ9
.
eyJpc3MiOiJqb2UiLA0KICJleHAiOjEzMDA4MTkzODAsDQogImh0dHA6Ly9leGFt
cGxlLnNvbS9pc19yb290Ijp0cnVlfQ
```

The resulting JWS Signing Input value, which is the ASCII representation of above string, is the following octet sequence:

```
[101, 121, 74, 104, 98, 71, 99, 105, 79, 105, 74, 70, 85, 122, 73,
49, 78, 105, 74, 57, 46, 101, 121, 74, 112, 99, 51, 77, 105, 79, 105,
74, 113, 98, 50, 85, 105, 76, 65, 48, 75, 73, 67, 74, 108, 101, 72,
65, 105, 79, 106, 69, 122, 77, 68, 65, 52, 77, 84, 107, 122, 79, 68,
65, 115, 68, 81, 111, 103, 73, 109, 104, 48, 100, 72, 65, 54, 76,
121, 57, 108, 101, 71, 70, 116, 99, 71, 120, 108, 76, 109, 78, 118,
98, 83, 57, 112, 99, 49, 57, 121, 98, 50, 57, 48, 73, 106, 112, 48,
99, 110, 86, 108, 102, 81]
```

This example uses the elliptic curve key represented in JSON Web Key [[JWK](#)] format below:

```
{"kty":"EC",
 "crv":"P-256",
 "x":"f830J3D2xF1Bg8vub9tLe1gHMzV76e8Tus9uPHvRVEU",
 "y":"x_FEzRu9m36HLN_tue659LNpXW6pCyStikYjKIWI5a0",
 "d":"jpsQnnGqML-YBIfFH1136cspYG6-0iY7X1fCE9-E9LI"
}
```


The ECDSA private part *d* is then passed to an ECDSA signing function, which also takes the curve type, P-256, the hash type, SHA-256, and the JWS Signing Input as inputs. The result of the digital signature is the EC point (*R*, *S*), where *R* and *S* are unsigned integers. In this example, the *R* and *S* values, given as octet sequences representing big endian integers are:

Result	Value
Name	
R	[14, 209, 33, 83, 121, 99, 108, 72, 60, 47, 127, 21, 88, 7, 212, 2, 163, 178, 40, 3, 58, 249, 124, 126, 23, 129, 154, 195, 22, 158, 166, 101]
S	[197, 10, 7, 211, 140, 60, 112, 229, 216, 241, 45, 175, 8, 74, 84, 128, 166, 101, 144, 197, 242, 147, 80, 154, 143, 63, 127, 138, 131, 163, 84, 213]

The JWS Signature is the value *R* || *S*. Encoding the signature as `BASE64URL(JWS Signature)` produces this value (with line breaks for display purposes only):

```
DtEhU3ljbEg8L38VWAFUAq0yKAM6-Xx-F4GawxaepmXFCgfTjDxw5djxLa8ISlSA
pmWQxfKTUJqPP3-Kg6NU1Q
```

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

```
eyJhbGciOiJIJFUiI1NiJ9
.
eyJpc3MiOiJqb2UiLA0KICJleHAiOjEzMDA4MTkzODAsDQogImh0dHA6Ly9leGFt
cGx1LmNvbS9pc19yb290Ijp0cnVlfQ
.
DtEhU3ljbEg8L38VWAFUAq0yKAM6-Xx-F4GawxaepmXFCgfTjDxw5djxLa8ISlSA
pmWQxfKTUJqPP3-Kg6NU1Q
```

[A.3.2.](#) Validating

Since the "alg" Header Parameter is "ES256", we validate the ECDSA P-256 SHA-256 digital signature contained in the JWS Signature.

Validating the JWS Signature is a bit different from the previous examples. We need to split the 64 member octet sequence of the JWS Signature (which is `base64url` decoded from the value encoded in the JWS representation) into two 32 octet sequences, the first

representing R and the second S. We then pass the public key (x, y), the signature (R, S), and the JWS Signing Input (which is the initial substring of the JWS Compact Serialization representation up until but not including the second period character) to an ECDSA signature verifier that has been configured to use the P-256 curve with the SHA-256 hash function.

[A.4.](#) Example JWS using ECDSA P-521 SHA-512

[A.4.1.](#) Encoding

The JWS Protected Header for this example differs from the previous example because different ECDSA curves and hash functions are used. The JWS Protected Header used is:

```
{"alg":"ES512"}
```

The octets representing UTF8(JWS Protected Header) in this example (using JSON array notation) are:

```
[123, 34, 97, 108, 103, 34, 58, 34, 69, 83, 53, 49, 50, 34, 125]
```

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

```
eyJhbGciOiJFUzUxMiJ9
```

The JWS Payload used in this example, is the ASCII string "Payload". The representation of this string is the octet sequence:

```
[80, 97, 121, 108, 111, 97, 100]
```

Encoding this JWS Payload as BASE64URL(JWS Payload) gives this value:

```
UGF5bG9hZA
```

Combining these as BASE64URL(UTF8(JWS Protected Header)) || '.' || BASE64URL(JWS Payload) gives this string:

```
eyJhbGciOiJFUzUxMiJ9.UGF5bG9hZA
```

The resulting JWS Signing Input value, which is the ASCII representation of above string, is the following octet sequence:

```
[101, 121, 74, 104, 98, 71, 99, 105, 79, 105, 74, 70, 85, 122, 85, 120, 77, 105, 74, 57, 46, 85, 71, 70, 53, 98, 71, 57, 104, 90, 65]
```

This example uses the elliptic curve key represented in JSON Web Key

[JWK] format below (with line breaks within values for display purposes only):

```
{
  "kty": "EC",
  "crv": "P-521",
  "x": "AekpBQ8ST8a8VcfV0TNl353vSrDCLLJXmPk06wTjxrrjcBpXp5E0nYG_
    NjFZ60vLFV1jSfS9tsz4qUxcWceqwQGk",
  "y": "ADSmRA43Z1DSNx_RvcLI87cdL07l6jQyyBXMoxVg_l2Th-x3S1WDhjDl
    y79ajL4Kkd0AZMaZmh9ubmf63e3kyMj2",
  "d": "AY5pb7A0UFiB3RELSd64fTLOSV_jazdF7fLYyuTw8l0fRhWg6Y6rUrPA
    xerEzgdRhajnu0ferB0d53vM9mE15j2C"
}
```

The ECDSA private part *d* is then passed to an ECDSA signing function, which also takes the curve type, P-521, the hash type, SHA-512, and the JWS Signing Input as inputs. The result of the digital signature is the EC point (*R*, *S*), where *R* and *S* are unsigned integers. In this example, the *R* and *S* values, given as octet sequences representing big endian integers are:

Result	Value
Name	
R	[1, 220, 12, 129, 231, 171, 194, 209, 232, 135, 233, 117, 247, 105, 122, 210, 26, 125, 192, 1, 217, 21, 82, 91, 45, 240, 255, 83, 19, 34, 239, 71, 48, 157, 147, 152, 105, 18, 53, 108, 163, 214, 68, 231, 62, 153, 150, 106, 194, 164, 246, 72, 143, 138, 24, 50, 129, 223, 133, 206, 209, 172, 63, 237, 119, 109]
S	[0, 111, 6, 105, 44, 5, 41, 208, 128, 61, 152, 40, 92, 61, 152, 4, 150, 66, 60, 69, 247, 196, 170, 81, 193, 199, 78, 59, 194, 169, 16, 124, 9, 143, 42, 142, 131, 48, 206, 238, 34, 175, 83, 203, 220, 159, 3, 107, 155, 22, 27, 73, 111, 68, 68, 21, 238, 144, 229, 232, 148, 188, 222, 59, 242, 103]

The JWS Signature is the value *R* || *S*. Encoding the signature as BASE64URL(JWS Signature) produces this value (with line breaks for display purposes only):

```
AdwMgeerwtHoh-l192l60hp9wAHZFVJbLfD_UxMi70cwnZOYaRI1bKPWR0c-mZZq
wqT2SI-KGDKB34X00aw_7XdtAG8GaSwFKdCAPZgoXD2YBJZCPEX3xKpRwcd008Kp
EHwJjyqOgzD07iKvU8vcnwNrmxYbSW9ERBXuk0XolLze0_Jn
```

Concatenating these values in the order Header.Payload.Signature with period ('.') characters between the parts yields this complete JWS

representation using the JWS Compact Serialization (with line breaks for display purposes only):

```
eyJhbGciOiJFUzUxMiJ9
.
UGF5bG9hZA
.
AdwMgeerwtHoh-l192l60hp9wAHZfVJbLfD_UxMi70cwnZOYaRI1bKPWR0c-mZZq
wqT2SI-KGDKB34X00aw_7XdtAG8GaSwFKdCAPZgoXD2YBJZCPEX3xKpRwcd008Kp
EHwJjyq0gzD07iKvU8vcnwNrmxYbSW9ERBXuk0XolLze0_Jn
```

[A.4.2.](#) Validating

Since the "alg" Header Parameter is "ES512", we validate the ECDSA P-521 SHA-512 digital signature contained in the JWS Signature.

Validating this JWS Signature is very similar to the previous example. We need to split the 132 member octet sequence of the JWS Signature into two 66 octet sequences, the first representing R and the second S. We then pass the public key (x, y), the signature (R, S), and the JWS Signing Input to an ECDSA signature verifier that has been configured to use the P-521 curve with the SHA-512 hash function.

[A.5.](#) Example Unsecured JWS

The following example JWS Protected Header declares that the encoded object is an Unsecured JWS:

```
{"alg":"none"}
```

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

```
eyJhbGciOiJub25lIn0
```

The JWS Payload used in this example, which follows, is the same as in the previous examples. Since the `BASE64URL(JWS Payload)` value will therefore be the same, its computation is not repeated here.

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

The JWS Signature is the empty octet string and `BASE64URL(JWS Signature)` is the empty string.

Concatenating these parts in the order `Header.Payload.Signature` with

period ('.') characters between the parts yields this complete JWS (with line breaks for display purposes only):

```
eyJhbGciOiJub251In0
.
eyJpc3MiOiJqb2UiLA0KICJleHAiOjEzMDA4MTkzODAsDQogImh0dHA6Ly9leGFt
cGx1LmNvbS9pc19yb290Ijp0cnVlfQ
.
```

[A.6.](#) Example JWS using General JWS JSON Serialization

This section contains an example using the general JWS JSON Serialization syntax. This example demonstrates the capability for conveying multiple digital signatures and/or MACs for the same payload.

The JWS Payload used in this example is the same as that used in the examples in [Appendix A.2](#) and [Appendix A.3](#) (with line breaks for display purposes only):

```
eyJpc3MiOiJqb2UiLA0KICJleHAiOjEzMDA4MTkzODAsDQogImh0dHA6Ly9leGFt
cGx1LmNvbS9pc19yb290Ijp0cnVlfQ
```

Two digital signatures are used in this example: the first using RSASSA-PKCS-v1_5 SHA-256 and the second using ECDSA P-256 SHA-256. For the first, the JWS Protected Header and key are the same as in [Appendix A.2](#), resulting in the same JWS Signature value; therefore, its computation is not repeated here. For the second, the JWS Protected Header and key are the same as in [Appendix A.3](#), resulting in the same JWS Signature value; therefore, its computation is not repeated here.

[A.6.1.](#) JWS Per-Signature Protected Headers

The JWS Protected Header value used for the first signature is:

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

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

```
eyJhbGciOiJSUzI1NiJ9
```

The JWS Protected Header value used for the second signature is:

```
{"alg":"ES256"}
```

Encoding this JWS Protected Header as BASE64URL(UTF8(JWS Protected

Header)) gives this value:

```
eyJhbGciOiJFUzI1NiJ9
```

[A.6.2.](#) JWS Per-Signature Unprotected Headers

Key ID values are supplied for both keys using per-signature Header Parameters. The two values used to represent these Key IDs are:

```
{"kid":"2010-12-29"}
```

and

```
{"kid":"e9bc097a-ce51-4036-9562-d2ade882db0d"}
```

[A.6.3.](#) Complete JOSE Header Values

Combining the protected and unprotected header values supplied, the JOSE Header values used for the first and second signatures respectively are:

```
{"alg":"RS256",  
 "kid":"2010-12-29"}
```

and

```
{"alg":"ES256",  
 "kid":"e9bc097a-ce51-4036-9562-d2ade882db0d"}
```

[A.6.4.](#) Complete JWS JSON Serialization Representation

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


```

{
  "payload":
    "eyJpc3MiOiJqb2UiLA0KICJleHAiOiEzMDA4MTkzODAsDQogImh0dHA6Ly9leGFtcGxlLmNvbS9pc19yb290Ijp0cnVlfQ",
  "signatures": [
    {
      "protected": "eyJhbGciOiJSUzI1NiJ9",
      "header":
        {
          "kid": "2010-12-29",
          "signature":
            "cC4hiUPoj9Eetdgtv3hF80EGrhuB__dzERat0XF9g2VtQgr9PJbu3X0iZj5RZmh7AAuHIm4Bh-0Qc_lF5YKt_08W2Fp5jujGbdS9uJdbF9CUAr7t1dnZcAcQjbKBYNX4BAynRFdiuB--f_nZLgrnbyTyWz075vRK5h6xBArLIARNPvkSjtQBMH1b1L07Qe7K0GarZRmB_eSN9383Lc0Ln6_d0--xi12jzDwusC-e0kHWEsqfFZESc6BfI7no0PqvhJ1phCnvWh6IeYI2w9Q0YEUipUTI8np6LbgGY9Fs98rqVt5AXLIhWkWyw1VmtVrBp0igcN_IoypGlUPQGe77Rw"},
      "protected": "eyJhbGciOiJFUzI1NiJ9",
      "header":
        {
          "kid": "e9bc097a-ce51-4036-9562-d2ade882db0d"},
      "signature":
        "DtEhU3ljBEG8L38VwAfUAq0yKAM6-Xx-F4GawxaepmXFCgfTjDxw5djxLa8ISlSApmWQxfKTUJqPP3-Kg6NU1Q"]
    }
  ]
}

```

A.7. Example JWS using Flattened JWS JSON Serialization

This section contains an example using the flattened JWS JSON Serialization syntax. This example demonstrates the capability for conveying a single digital signature or MAC in a flattened JSON structure.

The values in this example are the same as those in the second signature of the previous example in [Appendix A.6](#).

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

```

{
  "payload":
    "eyJpc3MiOiJqb2UiLA0KICJleHAiOiEzMDA4MTkzODAsDQogImh0dHA6Ly9leGFtcGxlLmNvbS9pc19yb290Ijp0cnVlfQ",
  "protected": "eyJhbGciOiJFUzI1NiJ9",
  "header":
    {
      "kid": "e9bc097a-ce51-4036-9562-d2ade882db0d"},
  "signature":
    "DtEhU3ljBEG8L38VwAfUAq0yKAM6-Xx-F4GawxaepmXFCgfTjDxw5djxLa8ISlSApmWQxfKTUJqPP3-Kg6NU1Q"
}

```


[Appendix B](#). "x5c" (X.509 Certificate Chain) Example

The JSON array below is an example of a certificate chain that could be used as the value of an "x5c" (X.509 Certificate Chain) Header Parameter, per [Section 4.1.6](#). Note that since these strings contain base64 encoded (not base64url encoded) values, they are allowed to contain white space and line breaks.

```
[ "MIIE3jCCA8agAwIBAgICAwEwDQYJKoZIhvcNAQEFBQAwwYzELMAkGA1UEBhMCVVMxITAFBgNVBAoTGFROZSBHbyBEYWRkeSBHcm91cCwgSW5jLjExMC8GA1UECXMor28gRGFkZHKgQ2xhc3MgMiBDZXJ0awZpY2F0aw9uIEF1dGhvcml0eTAeFw0wNjExMTYwMTU0MzdaFw0yNjExMTYwMTU0MzdaMIHKMQswCQYDVQQGEwJVUzEQMA4GA1UECBMHQXJpem9uYTETMBEGA1UEBxMKU2NvdHRzZGFsZTEaMBGGA1UEChMRR29EYWRkeS5jb20sIEluYy4xMzAxBgNVBAsTKmh0dHA6Ly9jZXJ0awZpY2F0ZXMuZ29kYWwRkeS5jb20vcmlvbm3NpdG9yeTEwMC4GA1UEAxMnR28gRGFkZHKgU2VjdXJlIENlc nRpZm1jYXRpb24gQXV0aG9yaXR5MREwDwYDVQQFEwgnZk20TI4NzCCASIwDQYJKoZIhvcNAQEBBQADggEPADCCAQoCggEBAMQt1RWMnCZM7DI161+4WQFapmGBWTt wY6vj3D3HkrjJM9N55DrtpDAjhI6zMBS2sofDPZVUBJ7fmd0LJR4h3mUpfjWoqV Tr9vcyOdQmVZwt7/v+WibXnvQAjYwqDL1CBM6nPWT27oDyqu9SoWlm2r4arV3aL GbqGmu75RpRSgAvSMeYddi5Kcju+GZtCpyz8/x4fKL4o/K1w/05epHBp+YlLpyo 7RJlbr2EkRtCDcVw5wrWCs9CHRK8r5RsL+H0EwnWGu1NcWdrxcx+AuP7q2BNgW JCJjP0q8lh8BJ6qf9Z/dFjpfMFDniNoW1fho3/Rb2cRGadDAW/hOUoz+EDU8CAw EAAaOCATiWggEuMB0GA1UdDgQWBbT9rGEyk2xF1uLuhV+auud2mWjM5zAfBgNVH SMEGDAWgBTSxLDSkdRMEXGzYcs9of7dqGrU4zASBgNVHRMBAf8ECDAGAQH/AgEA MDMGCCsGAQUFBwEBBCCwJTAjBggrBgEFBQcwAYYXaHR0cDovL29jc3AuZ29kYWwR keS5jb20wRgYDVR0fBD8wPTA7oDmgN4Y1aHR0cDovL2Nlc nRpZm1jYXRlc y5nb2RhZGR5LmNvbS9yZXBvc2l0b3J5L2dkcm9vdC5jcmwwSwYDVR0gBEQwQjBABgRVH SAAMDgwNgYIKwYBBQUHAgEwKmh0dHA6Ly9jZXJ0awZpY2F0ZXMuZ29kYWwRkeS5j b20vcmlvbm3NpdG9yeTA0BgNVHQ8BAf8EBAMCAQYwDQYJKoZIhvcNAQEFBQADggE BANKGwOy9+aG2Z+5mC6IG0gRQjhVyrEp0lVPLN8tESe8HkGsz2Zbw1Fa1EzAFPI UyIXvJxwqoJKSQ3kbTJSMUA2fCENZvD117esyfxVgqwcSeIaha86ykRvOe5GPLL 5CkKSkB2XiSkd83ASe8T+5o0yGPwLPk9Qnt0hCqU7S+8MxZC9Y7lhyVJEnfuz9 p0iRFEU00jZv2kwzRaJBdyTXRE4+uXR21aITVSzGh601mawGhId/dQb8vxRMDsx uxN89txJx90jxUUAiKEngHUuHqDTMBqLdElrRhjZkAzVvb3du6/KFUJheqwNTrZ EjYx8WnM25sgVjOuH0aBsXBTWVU+4=", "MIIE+zCCBGsgAwIBAgICAwEwDQYJKoZIhvcNAQEFBQAwwgbsxJDAiBgNVBACtG1Z hbG1dZXJ0IFZhbG1kYXRpb24gTmV0d29yazEXMBUGA1UEChMOVmFsaUNlc nQsIE luYy4xNTAzBgNVBAsTLFZhbG1dZXJ0IENsYXNzIDIGUG9saWN5IFZhbG1kYXRpb 24gQXV0aG9yaXR5MSEwHwYDVQQDExhodHRwOi8vd3d3LnZhbmG1jZXJ0LmNvbS8x IDAeBgkqhkiG9w0BCQEWEluZm9AdmFsaWNlc nQuY29tMB4XDTA0MDYyOTE3MDY yMFoXDTI0MDYyOTE3MDYyMFowYzELMAkGA1UEBhMCVVMxITAFBgNVBAoTGFROZS BHbyBEYWRkeSBHcm91cCwgSW5jLjExMC8GA1UECXMor28gRGFkZHKgQ2xhc3MgM iBDZXJ0awZpY2F0aw9uIEF1dGhvcml0eTCCASAwDQYJKoZIhvcNAQEBBQADggEN ADCCAQgCggEBAN6d1+pXGEmhw+vXX0iG6r7d/+TvZxz0ZWizV3GgXne77ZtJ6XC APVYYYwhv2vLM0D9/AlQiVBDYsoHUuHU9S3/Hd8M+eKsaA7Ugay9qK7HFih7Eux 6wwdhfJ2+qn1j3hybX2C32qRe3H3I2TqYXP2WYktsqbl2i/ojgC95/5Y0V4evL0 tXiEqITLdiOr18SPaAIBQi2XKV10ARFmR6jYGB0xUGlcmIbYsUfb18aQr4CUWwo riMYavx4A61NF4DD+qta/KFApMoZFv6yy09ecw3ud72a9nmYvLEHZ6IVDd2gWMZ
```


Eewo+YihfukEHU1jPEX44dMX4/7VpkI+Ed0qXG68CAQ0jggHhMIIB3TAdBgNVHQ
4EFgQU0sSw0pHUTBFxs2HLPaH+3ahq10MwgdIGA1UdIwSBYjCBx6GBwaSBvjCBu
zEkMCIGA1UEBxMbVmFsaUNlcnQgVmFsaWRhdGlviB0ZXR3b3JrMRcwFQYDVQK
Ew5WYwXpQ2VydCwgSW5jLjE1MDMGA1UECXMVMFsaUNlcnQgQ2xhc3MgMiBQb2x
pY3kgVmFsaWRhdGlviBBdXRob3JpdHkxITAFBgNVBAMTGGh0dHA6Ly93d3cudm
FsaWNlcnQuY29tLzEgMB4GCSqGSIB3DQEJARYRaW5mb0B2YwXpY2VydC5jb22CA
QEwDwYDVR0TAQH/BAUwAwEB/zAzBggrBgEFBQcBAQQnMCUwIwYIKwYBBQUHMAGG
F2h0dHA6Ly9vY3NwLmdvZGFkZHKuY29tMEQGA1UdHwQ9MDswOaA3oDWGM2h0dHA
6Ly9jZXJ0aWZpY2F0ZXMuZ29kYWRkeS5jb20vcmlvbn3NpdG9yeS9yb290LmNybd
BLBgNVHSAERDBCEAGBFudIAAw0DA2BggrBgEFBQcCARYqaHR0cDovL2NlcnRpZ
mljYXRlcY5nb2RrZGR5LmNvbS9yZXBvc2l0b3J5MA4GA1UdDwEB/wQEAwIBBjAN
BgkqhkiG9w0BAQUFAA0BgQC1QPMnHfbq/qQaQlpE9xXUuUaJwL6e4+PrxeNYiY+
Sn1eocSxI0YgYeR+sBjUZsE40WBsUs5iB0QQeyAfJg594RAoYC5jcdnplDQ1tgM
QLARzLrUc+cb53S8wGd9D0VmsfSx0aFIqII6hR8INMqzW/Rn453HWkrugp++85j
09VZw==",
"MIIC5zCCA1ACAQEwDQYJKoZIhvcNAQEFBQAwbgsxJDAiBgNVBACtG1ZhbG1DZXJ
0IFZhbGlkYXRpb24gTmV0d29yazEXMBUGA1UEChMOVmFsaUNlcnQsIEluYy4xNT
AzBgNVBAsTLFZhbG1DZXJ0IENsYXNzIDIGUG9saWN5IFZhbGlkYXRpb24gQXV0a
G9yaXR5MSEwHwYDVQQDEExhbm90b3R3d3d3LnZhbGljZXJ0LmNvbS8xIDAeBgkq
hkiG9w0BCQEWEluZm9AdmFsaWNlcnQuY29tMB4XDTk5MDYyNjAwMTk1NFoXDTE
5MDYyNjAwMTk1NFowgbsxJDAiBgNVBACtG1ZhbG1DZXJ0IFZhbGlkYXRpb24gTm
V0d29yazEXMBUGA1UEChMOVmFsaUNlcnQsIEluYy4xNTAzBgNVBAsTLFZhbG1DZ
XJ0IENsYXNzIDIGUG9saWN5IFZhbGlkYXRpb24gQXV0aG9yaXR5MSEwHwYDVQQD
Exhbm90b3R3d3d3LnZhbGljZXJ0LmNvbS8xIDAeBgkqhkiG9w0BCQEWEluZm9
AdmFsaWNlcnQuY29tMIGfMA0GCSqGSIB3DQEBAQUAA4GNADCBiQKBgQD00nHK5a
vIWZJV16vYdA757tn2VUdZZUc0BVXc65g2PFxTXdMwzzjsvUGJ7SVCCSRrC16zf
N1SLUzm1NZ9WlmpZdRJEy0kTRxQb7XBhVQ7/nHk01xC+YDgkRoKwzk2Z/M/VXwb
P7RfZHM047QSV4dk+NoS/zcnwbNDu+97bi5p9WIDAQABMA0GCSqGSIB3DQEBBQU
AA4GBADt/UG9vUJSZSWI40B9L+KXIPqeCgfYrx+jFzug6EILLGAC0Tb2oWH+heQ
C1u+mNr0HZDzTuIYEZoDJJKPTEjlbVUjP9UNV+mWwD5MlM/Mtsq2azSiGM5bUMM
j4QssxsodyamEwCW/P0uZ6lcg5Ktz885hZo+L7tdEy8W9ViH0Pd"]

[Appendix C.](#) Notes on implementing base64url encoding without padding

This appendix describes how to implement base64url encoding and decoding functions without padding based upon standard base64 encoding and decoding functions that do use padding.

To be concrete, example C# code implementing these functions is shown below. Similar code could be used in other languages.


```
static string base64urlencode(byte [] arg)
{
    string s = Convert.ToBase64String(arg); // Regular base64 encoder
    s = s.Split('=')[0]; // Remove any trailing '='s
    s = s.Replace('+', '-'); // 62nd char of encoding
    s = s.Replace('/', '_'); // 63rd char of encoding
    return s;
}

static byte [] base64urldecode(string arg)
{
    string s = arg;
    s = s.Replace('-', '+'); // 62nd char of encoding
    s = s.Replace('_', '/'); // 63rd char of encoding
    switch (s.Length % 4) // Pad with trailing '='s
    {
        case 0: break; // No pad chars in this case
        case 2: s += "=="; break; // Two pad chars
        case 3: s += "="; break; // One pad char
        default: throw new System.Exception(
            "Illegal base64url string!");
    }
    return Convert.FromBase64String(s); // Standard base64 decoder
}
```

As per the example code above, the number of '=' padding characters that needs to be added to the end of a base64url encoded string without padding to turn it into one with padding is a deterministic function of the length of the encoded string. Specifically, if the length mod 4 is 0, no padding is added; if the length mod 4 is 2, two '=' padding characters are added; if the length mod 4 is 3, one '=' padding character is added; if the length mod 4 is 1, the input is malformed.

An example correspondence between unencoded and encoded values follows. The octet sequence below encodes into the string below, which when decoded, reproduces the octet sequence.

3 236 255 224 193

A-z_4ME

[Appendix D](#). Notes on Key Selection

This appendix describes a set of possible algorithms for selecting the key to be used to validate the digital signature or MAC of a JWS object or for selecting the key to be used to decrypt a JWE object. This guidance describes a family of possible algorithms, rather than a single algorithm, because in different contexts, not all the

sources of keys will be used, they can be tried in different orders, and sometimes not all the collected keys will be tried; hence, different algorithms will be used in different application contexts.

The steps below are described for illustration purposes only; specific applications can and are likely to use different algorithms or perform some of the steps in different orders. Specific applications will frequently have a much simpler method of determining the keys to use, as there may be one or two key selection methods that are profiled for the application's use. This appendix supplements the normative information on key location in [Section 6](#).

These algorithms include the following steps. Note that the steps can be performed in any order and do not need to be treated as distinct. For example, keys can be tried as soon as they are found, rather than collecting all the keys before trying any.

1. Collect the set of potentially applicable keys. Sources of keys may include:
 - * Keys supplied by the application protocol being used.
 - * Keys referenced by the "jku" (JWK Set URL) Header Parameter.
 - * The key provided by the "jwk" (JSON Web Key) Header Parameter.
 - * The key referenced by the "x5u" (X.509 URL) Header Parameter.
 - * The key provided by the "x5c" (X.509 Certificate Chain) Header Parameter.
 - * Other applicable keys available to the application.

The order for collecting and trying keys from different key sources is typically application dependent. For example, frequently all keys from a one set of locations, such as local caches, will be tried before collecting and trying keys from other locations.

2. Filter the set of collected keys. For instance, some applications will use only keys referenced by "kid" (key ID) or "x5t" (X.509 certificate SHA-1 thumbprint) parameters. If the application uses the "alg" (algorithm), "use" (public key use), or "key_ops" (key operations) parameters, keys with inappropriate values of those parameters would be excluded. Additionally, keys might be filtered to include or exclude keys with certain other member values in an application specific manner. For some applications, no filtering will be applied.

3. Order the set of collected keys. For instance, keys referenced by "kid" (Key ID) or "x5t" (X.509 Certificate SHA-1 Thumbprint) parameters might be tried before keys with neither of these values. Likewise, keys with certain member values might be ordered before keys with other member values. For some applications, no ordering will be applied.
4. Make trust decisions about the keys. Signatures made with keys not meeting the application's trust criteria would not be accepted. Such criteria might include, but is not limited to the source of the key, whether the TLS certificate validates for keys retrieved from URLs, whether a key in an X.509 certificate is backed by a valid certificate chain, and other information known by the application.
5. Attempt signature or MAC validation for a JWS object or decryption of a JWE object with some or all of the collected and possibly filtered and/or ordered keys. A limit on the number of keys to be tried might be applied. This process will normally terminate following a successful validation or decryption.

Note that it is reasonable for some applications to perform signature or MAC validation prior to making a trust decision about a key, since keys for which the validation fails need no trust decision.

[Appendix E](#). Negative Test Case for "crit" Header Parameter

Conforming implementations must reject input containing critical extensions that are not understood or cannot be processed. The following JWS must be rejected by all implementations, because it uses an extension Header Parameter name "http://example.invalid/UNDEFINED" that they do not understand. Any other similar input, in which the use of the value "http://example.invalid/UNDEFINED" is substituted for any other Header Parameter name not understood by the implementation, must also be rejected.

The JWS Protected Header value for this JWS is:

```
{
  "alg": "none",
  "crit": ["http://example.invalid/UNDEFINED"],
  "http://example.invalid/UNDEFINED": true
}
```


The complete JWS that must be rejected is as follows (with line breaks for display purposes only):

```
eyJhbGciOiJub251IiwNCiAiY3JpdCI6WyJodHRwOi8vZXhhbXBsZS5jb20vVU5ERU
ZJTkVEIl0sDQogImh0dHA6Ly9leGFtcGxlLnNvbS9VTkrFRkl0RUQiOnRydWUNCn0.
RkFJTA.
```

[Appendix F.](#) Detached Content

In some contexts, it is useful integrity protect content that is not itself contained in a JWS object. One way to do this is create a JWS object in the normal fashion using a representation of the content as the payload, but then delete the payload representation from the JWS, and send this modified object to the recipient, rather than the JWS. When using the JWS Compact Serialization, the deletion is accomplished by replacing the second field (which contains `BASE64URL(JWS Payload)`) value with the empty string; when using the JWS JSON Serialization, the deletion is accomplished by deleting the "payload" member. This method assumes that the recipient can reconstruct the exact payload used in the JWS. To use the modified object, the recipient reconstructs the JWS by re-inserting the payload representation into the modified object, and uses the resulting JWS in the usual manner. Note that this method needs no support from JWS libraries, as applications can use this method by modifying the inputs and outputs of standard JWS libraries.

[Appendix G.](#) Acknowledgements

Solutions for signing JSON content were previously explored by Magic Signatures [[MagicSignatures](#)], JSON Simple Sign [[JSS](#)], and Canvas Applications [[CanvasApp](#)], all of which influenced this draft.

Thanks to Axel Nennker for his early implementation and feedback on the JWS and JWE specifications.

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

Dirk Balfanz, Richard Barnes, Brian Campbell, Alissa Cooper, Breno de Medeiros, Stephen Farrell, Dick Hardt, Joe Hildebrand, Jeff Hodges, Russ Housley, Edmund Jay, Tero Kivinen, Yaron Y. Goland, Ben Laurie, Ted Lemon, James Manger, Matt Miller, Kathleen Moriarty, Tony Nadalin, Hideki Nara, Axel Nennker, John Panzer, Ray Polk, Emmanuel Raviart, Eric Rescorla, Pete Resnick, Jim Schaad, Paul Tarjan, Hannes

Tschafenig, and Sean Turner.

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

Appendix H. Document History

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

-36

- o Defined a flattened JWS JSON Serialization syntax, which is optimized for the single digital signature or MAC case.
- o Clarified where white space and line breaks may occur in JSON objects by referencing [Section 2 of RFC 7159](#).
- o Specified that registration reviews occur on the jose-reg-review@ietf.org mailing list.

-35

- o Addressed AppsDir reviews by Ray Polk.
- o Used real values for examples in the IANA Registration Template.

-34

- o Addressed IESG review comments by Alissa Cooper, Pete Resnick, Richard Barnes, Ted Lemon, and Stephen Farrell.
- o Addressed Gen-ART review comments by Russ Housley.
- o Referenced [RFC 4945](#) for PEM certificate delimiter syntax.

-33

- o Noted that certificate thumbprints are also sometimes known as certificate fingerprints.
- o Added an informative reference to [draft-ietf-uta-tls-bcp](#) for recommendations on improving the security of software and services using TLS.
- o Changed the registration review period to three weeks.

- o Acknowledged additional contributors.

-32

- o Addressed Gen-ART review comments by Russ Housley.
- o Addressed secdir review comments by Tero Kivinen, Stephen Kent, and Scott Kelly.
- o Replaced the term Plaintext JWS with Unsecured JWS.

-31

- o Reworded the language about JWS implementations ignoring the "typ" and "cty" parameters, explicitly saying that their processing is performed by JWS applications.
- o Added additional guidance on ciphersuites currently considered to be appropriate for use, including a reference to a recent update by the TLS working group.

-30

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

-29

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

-28

- o Revised the introduction to the Security Considerations section. Also introduced additional subsection headings for security considerations items and also moved a security consideration item here from the JWA draft.
- o Added text about when applications typically would and would not use "typ" and "cty" header parameters.

-27

- o Added the "x5t#S256" (X.509 Certificate SHA-256 Thumbprint) header parameter.
- o Stated that any JSON inputs not conforming to the JSON-text syntax defined in [RFC 7159](#) input MUST be rejected in their entirety.
- o Simplified the TLS requirements.

-26

- o Referenced [Section 6 of RFC 6125](#) for TLS server certificate identity validation.
- o Described potential sources of ambiguity in representing the JSON objects used in the examples. The octets of the actual UTF-8 representations of the JSON objects used in the examples are included to remove these ambiguities.
- o Added a small amount of additional explanatory text to the signature validation examples to aid implementers.
- o Noted that octet sequences are depicted using JSON array notation.
- o Updated references, including to W3C specifications.

-25

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

-24

- o Updated the JSON reference to [RFC 7159](#).

-23

- o Clarified that the base64url encoding includes no line breaks, white space, or other additional characters.

-22

- o Corrected [RFC 2119](#) terminology usage.
- o Replaced references to [draft-ietf-json-rfc4627bis](#) with [RFC 7158](#).

-21

- o Applied review comments to the appendix "Notes on Key Selection", addressing issue #93.
- o Changed some references from being normative to informative, addressing issue #90.
- o Applied review comments to the JSON Serialization section, addressing issue #121.

-20

- o Made terminology definitions more consistent, addressing issue #165.
- o Restructured the JSON Serialization section to call out the parameters used in hanging lists, addressing issue #121.
- o Described key filtering and refined other aspects of the text in the appendix "Notes on Key Selection", addressing issue #93.
- o Replaced references to [RFC 4627](#) with [draft-ietf-json-rfc4627bis](#), addressing issue #90.

-19

- o Added the appendix "Notes on Validation Key Selection", addressing issue #93.
- o Reordered the key selection parameters.

-18

- o Updated the mandatory-to-implement (MTI) language to say that applications using this specification need to specify what serialization and serialization features are used for that application, addressing issue #119.
- o Changes to address editorial and minor issues #25, #89, #97, #110, #114, #115, #116, #117, #120, and #184.
- o Added and used Header Parameter Description registry field.

-17

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

- o Updated the mandatory-to-implement (MTI) language to say that general-purpose implementations must implement the single signature/MAC value case for both serializations whereas special-purpose implementations can implement just one serialization if that meets the needs of the use cases the implementation is designed for, addressing issue #119.
- o Explicitly named all the logical components of a JWS and defined the processing rules and serializations in terms of those components, addressing issues #60, #61, and #62.
- o Replaced verbose repetitive phrases such as "base64url encode the octets of the UTF-8 representation of X" with mathematical notation such as "BASE64URL(UTF8(X))".
- o Terms used in multiple documents are now defined in one place and incorporated by reference. Some lightly used or obvious terms were also removed. This addresses issue #58.

-16

- o Changes to address editorial and minor issues #50, #98, #99, #102, #104, #106, #107, #111, and #112.

-15

- o Clarified that it is an application decision which signatures, MACs, or plaintext values must successfully validate for the JWS to be accepted, addressing issue #35.
- o Corrected editorial error in "ES512" example.
- o Changes to address editorial and minor issues #34, #96, #100, #101, #104, #105, and #106.

-14

- o Stated that the "signature" parameter is to be omitted in the JWS JSON Serialization when its value would be empty (which is only the case for a Plaintext JWS).

-13

- o Made all header parameter values be per-signature/MAC, addressing issue #24.

-12

- o Clarified that the "typ" and "cty" header parameters are used in an application-specific manner and have no effect upon the JWS processing.
- o Replaced the MIME types "application/jws+json" and "application/jws" with "application/jose+json" and "application/jose".
- o Stated that recipients MUST either reject JWSs with duplicate Header Parameter Names or use a JSON parser that returns only the lexically last duplicate member name.
- o Added a Serializations section with parallel treatment of the JWS Compact Serialization and the JWS JSON Serialization and also moved the former Implementation Considerations content there.

-11

- o Added Key Identification section.
- o For the JWS JSON Serialization, enable header parameter values to be specified in any of three parameters: the "protected" member that is integrity protected and shared among all recipients, the "unprotected" member that is not integrity protected and shared among all recipients, and the "header" member that is not integrity protected and specific to a particular recipient. (This does not affect the JWS Compact Serialization, in which all header parameter values are in a single integrity protected JWE Header value.)
- o Removed suggested compact serialization for multiple digital signatures and/or MACs.
- o Changed the MIME type name "application/jws-js" to "application/jws+json", addressing issue #22.
- o Tightened the description of the "crit" (critical) header parameter.
- o Added a negative test case for the "crit" header parameter

-10

- o Added an appendix suggesting a possible compact serialization for JWSs with multiple digital signatures and/or MACs.

-09

- o Added JWS JSON Serialization, as specified by [draft-jones-jose-jws-json-serialization-04](#).
- o Registered "application/jws-js" MIME type and "JWS-JS" typ header parameter value.
- o Defined that the default action for header parameters that are not understood is to ignore them unless specifically designated as "MUST be understood" or included in the new "crit" (critical) header parameter list. This addressed issue #6.
- o Changed term "JWS Secured Input" to "JWS Signing Input".
- o Changed from using the term "byte" to "octet" when referring to 8 bit values.
- o Changed member name from "recipients" to "signatures" in the JWS JSON Serialization.
- o Added complete values using the JWS Compact Serialization for all examples.

-08

- o Applied editorial improvements suggested by Jeff Hodges and Hannes Tschofenig. Many of these simplified the terminology used.
- o Clarified statements of the form "This header parameter is OPTIONAL" to "Use of this header parameter is OPTIONAL".
- o Added a Header Parameter Usage Location(s) field to the IANA JSON Web Signature and Encryption Header Parameters registry.
- o Added seriesInfo information to Internet Draft references.

-07

- o Updated references.

-06

- o Changed "x5c" (X.509 Certificate Chain) representation from being a single string to being an array of strings, each containing a single base64 encoded DER certificate value, representing elements of the certificate chain.
- o Applied changes made by the RFC Editor to [RFC 6749](#)'s registry language to this specification.

-05

- o Added statement that "StringOrURI values are compared as case-sensitive strings with no transformations or canonicalizations applied".
- o Indented artwork elements to better distinguish them from the body text.

-04

- o Completed JSON Security Considerations section, including considerations about rejecting input with duplicate member names.
- o Completed security considerations on the use of a SHA-1 hash when computing "x5t" (x.509 certificate thumbprint) values.
- o Refer to the registries as the primary sources of defined values and then secondarily reference the sections defining the initial contents of the registries.
- o Normatively reference XML DSIG 2.0 for its security considerations.
- o Added this language to Registration Templates: "This name is case sensitive. Names that match other registered names in a case insensitive manner SHOULD NOT be accepted."
- o Reference [draft-jones-jose-jws-json-serialization](#) instead of [draft-jones-json-web-signature-json-serialization](#).
- o Described additional open issues.
- o Applied editorial suggestions.

-03

- o Added the "cty" (content type) header parameter for declaring type information about the secured content, as opposed to the "typ" (type) header parameter, which declares type information about this object.
- o Added "Collision Resistant Namespace" to the terminology section.
- o Reference ITU.X690.1994 for DER encoding.
- o Added an example JWS using ECDSA P-521 SHA-512. This has particular illustrative value because of the use of the 521 bit

integers in the key and signature values. This is also an example in which the payload is not a base64url encoded JSON object.

- o Added an example "x5c" value.
- o No longer say "the UTF-8 representation of the JWS Secured Input (which is the same as the ASCII representation)". Just call it "the ASCII representation of the JWS Secured Input".
- o Added Registration Template sections for defined registries.
- o Added Registry Contents sections to populate registry values.
- o Changed name of the JSON Web Signature and Encryption "typ" Values registry to be the JSON Web Signature and Encryption Type Values registry, since it is used for more than just values of the "typ" parameter.
- o Moved registries JSON Web Signature and Encryption Header Parameters and JSON Web Signature and Encryption Type Values to the JWS specification.
- o Numerous editorial improvements.

-02

- o Clarified that it is an error when a "kid" value is included and no matching key is found.
- o Removed assumption that "kid" (key ID) can only refer to an asymmetric key.
- o Clarified that JWSs with duplicate Header Parameter Names MUST be rejected.
- o Clarified the relationship between "typ" header parameter values and MIME types.
- o Registered application/jws MIME type and "JWS" typ header parameter value.
- o Simplified JWK terminology to get replace the "JWK Key Object" and "JWK Container Object" terms with simply "JSON Web Key (JWK)" and "JSON Web Key Set (JWK Set)" and to eliminate potential confusion between single keys and sets of keys. As part of this change, the Header Parameter Name for a public key value was changed from "jpk" (JSON Public Key) to "jwk" (JSON Web Key).

- o Added suggestion on defining additional header parameters such as "x5t#S256" in the future for certificate thumbprints using hash algorithms other than SHA-1.
- o Specify [RFC 2818](#) server identity validation, rather than [RFC 6125](#) (paralleling the same decision in the OAuth specs).
- o Generalized language to refer to Message Authentication Codes (MACs) rather than Hash-based Message Authentication Codes (HMACs) unless in a context specific to HMAC algorithms.
- o Reformatted to give each header parameter its own section heading.

-01

- o Moved definition of Plaintext JWSs (using "alg":"none") here from the JWT specification since this functionality is likely to be useful in more contexts than just for JWTs.
- o Added "jpk" and "x5c" header parameters for including JWK public keys and X.509 certificate chains directly in the header.
- o Clarified that this specification is defining the JWS Compact Serialization. Referenced the new JWS-JS spec, which defines the JWS JSON Serialization.
- o Added text "New header parameters should be introduced sparingly since an implementation that does not understand a parameter MUST reject the JWS".
- o Clarified that the order of the creation and validation steps is not significant in cases where there are no dependencies between the inputs and outputs of the steps.
- o Changed "no canonicalization is performed" to "no canonicalization need be performed".
- o Corrected the Magic Signatures reference.
- o Made other editorial improvements suggested by JOSE working group participants.

-00

- o Created the initial IETF draft based upon [draft-jones-json-web-signature-04](#) with no normative changes.

- o Changed terminology to no longer call both digital signatures and HMACs "signatures".

Authors' Addresses

Michael B. Jones
Microsoft

Email: mbj@microsoft.com

URI: <http://self-issued.info/>

John Bradley
Ping Identity

Email: ve7jtb@ve7jtb.com

URI: <http://www.thread-safe.com/>

Nat Sakimura
Nomura Research Institute

Email: n-sakimura@nri.co.jp

URI: <http://nat.sakimura.org/>

