TOC

JSON Web Signature (JWS)
draft-ietf-jose-json-web-signature-20

Abstract

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

Status of this Memo

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

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

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

This Internet-Draft will expire on July 24, 2014.

Copyright Notice

Table of Contents

TOC

1. Introduction

JSON Web Signature (JWS) represents content secured with digital signatures or Message Authentication Codes (MACs) using JavaScript Object Notation (JSON)

[I-D.ietf-json-rfc4627bis] based data structures. The JWS cryptographic mechanisms provide integrity protection for an arbitrary sequence of octets.

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", "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 **[USASCII]** representation of STRING.

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

---

2. Terminology

JSON Web Signature (JWS)
> A data structure representing a digitally signed or MACed message.

JWS Header
> JSON object containing the parameters describing the cryptographic operations and parameters employed. The JWS Header members are the union of the members of the JWS Protected Header and the JWS Unprotected Header. The members of the JWS Header are 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 JWS Header.

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

JWS Unprotected Header
> JSON object that contains the JWS 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). (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.

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.

3. JSON Web Signature (JWS) Overview

JWS represents digitally signed or MACed content using JSON data structures and base64url
encoding. A JWS represents these logical values:

JWS Header
    JSON object containing the parameters describing the cryptographic operations
    and parameters employed. The JWS Header members are the union of the
    members of the JWS Protected Header and the JWS Unprotected Header, as
    described below.

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.

The JWS Header represents the combination of these values:

JWS Protected Header
    JSON object that contains the JWS Header Parameters that are integrity protected
    by the JWS Signature digital signature or MAC operation.

JWS Unprotected Header
    JSON object that contains the JWS Header Parameters that are not integrity
    protected.

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 for transmission, since JSON lacks a way to directly
represent octet sequences.

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

In the JWS Compact Serialization, a JWS object is represented as the combination of these
three string values,

    BASE64URL(UTF8(JWS Protected Header)),

    BASE64URL(JWS Payload), and

    BASE64URL(JWS Signature),

concatenated in that order, with the three strings being separated by two period ('.') characters.

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 JWS Header are the combination 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), and

BASE64URL(JWS Signature),

with the three base64url encoding result strings and the JWS Unprotected Header value being 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.

3.1. 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 algorithm:

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

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

```
eyJ0eXAiOiJKV1QiLA0KICJhbGciOiJIUzI1NiJ9
```

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

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

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

```
eyJpc3MiOiJqb2UiLA0KICJleHAiOjEzMDA4MTkzODAsDQogImh0dHA6Ly9leGFt
cGxlLmNvbS9pc19yb290Ijp0cnVlfQ
```

Computing the HMAC of the JWS Signing Input ASCII(BASE64URL(UTF8(JWS Protected Header)) || '.' || BASE64URL(JWS Payload)) with the HMAC SHA-256 algorithm using the key

specified in **Appendix A.1** and base64url encoding the result yields this BASE64URL(JWS Signature) value:

```
dBjftJeZ4CVP-mB92K27uhbUJU1p1r_wW1gFWFOEjXk
```

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

```
eyJ0eXAiOiJKV1QiLA0KICJhbGciOiJIUzI1NiJ9
.
eyJpc3MiOiJqb2UiLA0KICJleHAiOjEzMDA4MTkzODAsDQogImh0dHA6Ly9leGFt
cGxlLmNvbS9pc19yb290Ijp0cnVlfQ
.
dBjftJeZ4CVP-mB92K27uhbUJU1p1r_wW1gFWFOEjXk
```

See **Appendix A** for additional examples.

---

4. JWS Header

The members of the JSON object(s) representing the JWS 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 JWS Header MUST be unique; recipients 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 4.1.10**, 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 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 signature, MAC, or plaintext 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 3.1 of HTTP Over TLS **[RFC2818]**. 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 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 **[RFC1421]**. 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 3.1 of HTTP Over TLS **[RFC2818]**. 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 **[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. Use of this Header Parameter is OPTIONAL.

If, in the future, certificate thumbprints need to be computed using hash functions other than SHA-1, it is suggested that additional related Header Parameters be defined for that purpose. For example, it is suggested that a new `x5t#S256` (X.509 Certificate Thumbprint using SHA-256) Header Parameter could be defined by registering it in the IANA JSON Web Signature and Encryption Header Parameters registry defined in **Section 9.1**.

4.1.8. "typ" (Type) Header Parameter

The `typ` (type) Header Parameter is used to declare the MIME Media Type **[IANA.MediaTypes]** of this complete JWS object in contexts where this is useful to the application. This parameter has no effect upon the JWS processing. Use of this Header Parameter is OPTIONAL.

Per **[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 senders 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.

4.1.9. "cty" (Content Type) Header Parameter

The `cty` (content type) Header Parameter is used to declare the MIME Media Type **[IANA.MediaTypes]** of the secured content (the payload) in contexts where this is useful to the application. This parameter has no effect upon the JWS processing. Use of this Header Parameter is OPTIONAL.

Per **[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 senders 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"`.

4.1.10. "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 JWS Header that use those extensions. If any of the listed extension Header Parameters are not understood and supported by the receiver, it MUST reject the JWS. Senders 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 JWS Header in the `crit` list. Senders 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. This Header Parameter MUST be integrity protected, and therefore MUST occur only within the JWS Protected Header, when used. 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 JWS Header: the JWS Protected Header, and if the JWS JSON Serialization is being used, 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 JWS 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. These three encoded values are used in both the JWS Compact Serialization and the JWS JSON Serialization representations.
8. If the JWS JSON Serialization is being used, repeat this process (steps 3-7) for each digital signature or MAC operation being performed.
9. 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.

It is an application decision which signatures, MACs, or plaintext 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 signature, MAC, or plaintext value needs to be successfully validated. However, in all cases, at least one signature, MAC, or plaintext 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, the base64url encoded representations of the JWS Protected Header, the JWS Payload, and the JWS Signature, and when using the JWS JSON Serialization, also 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, separated by two period ('.') characters. The JWS JSON Serialization is described in **Section 7.2**.
2. The encoded representation of the JWS Protected Header MUST be successfully base64url decoded following the restriction that no padding characters have been used.
3. The resulting octet sequence MUST be a UTF-8 encoded representation of a completely valid JSON object conforming to **[I-D.ietf-json-rfc4627bis]**, which is the JWS Protected Header.
4. If using the JWS Compact Serialization, let the JWS Header be the JWS Protected Header; otherwise, when using the JWS JSON Serialization, let the JWS 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.
5. The resulting JWS Header MUST 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 JWS Header.
6. 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.

7. The encoded representation of the JWS Payload MUST be successfully base64url decoded following the restriction that no padding characters have been used.

8. The encoded representation of the JWS Signature MUST be successfully base64url decoded following the restriction that no padding characters have been used.

9. The JWS Signature MUST be successfully validated 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.

10. If the JWS JSON Serialization is being used, repeat this process (steps 4-9) for each digital signature or MAC value contained in the representation.

5.3. String Comparison Rules

Processing a JWS inevitably requires comparing known strings to members and values in a JSON object. For example, in checking what the algorithm is, the Unicode string `alg` will be checked against the member names in the JWS 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.

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. The JSON rules for doing member name comparison are described in Section 8.3 of **[I-D.ietf-json-rfc4627bis]**.

Also, see the JSON security considerations in **Section 10.2** and the Unicode security considerations in **Section 10.3**.

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`, and `x5t` 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 sender 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. Content using the JWS JSON Serialization can be secured with more than one digital signature and/or MAC operation. This representation is neither optimized for compactness nor URL-safe.

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

payload
> The value BASE64URL(JWS Payload) is stored in the `payload` member.

signatures
> A JSON array in the `signatures` member is used to hold values that are specific to a particular signature or MAC computation, with one array element per signature/MAC represented. These array elements are JSON objects, as specified below.

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

protected
> Each value BASE64URL(UTF8(JWS Protected Header)), if non-empty, is stored in the `protected` member.

header
> Each JWS Unprotected Header value, if non-empty, is stored in the `header` member. If present, a JWS Unprotected Header value is represented as an unencoded JSON object, rather than as a string.

signature
> Each value BASE64URL(JWS Signature) is stored in the `signature` member.

Of these members of the two JSON objects defined above, only the `payload`, `signatures`, and `signature` members MUST be present. 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 values represented in the `protected` member of the signature/MAC's array element, and (2) the JWS Unprotected Header values in the `header` member of the signature/MAC's array element. The union of these sets of Header Parameters comprises the JWS Header. The Header Parameter names in the two locations MUST be disjoint.

The contents of the JWS Payload and JWS Signature values are exactly as defined in the rest of this specification. They are interpreted and validated in the same manner, with each corresponding JWS Signature and set of Header Parameter values being created and validated together.

Each JWS Signature value is computed on the JWS Signing Input using the parameters of the corresponding JWS 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 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 of computing a JWS using the JWS JSON Serialization.

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, but has very limited actual deployment, and might not be readily available in implementation toolkits. TLS version 1.0 **[RFC2246]** is the most widely deployed version, and will give the broadest interoperability.

To protect against information disclosure and tampering, confidentiality protection MUST be applied using TLS with a ciphersuite that provides confidentiality and integrity protection.

Whenever TLS is used, a TLS server certificate check MUST be performed, per **RFC 6125** [RFC6125].

9. IANA Considerations

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

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

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

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

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).

9.1. JSON Web Signature and Encryption Header Parameters Registry

This specification establishes the IANA JSON Web Signature and Encryption Header Parameters registry for JWS and JWE 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., "example"). 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., "Example description").

Header Parameter Usage Location(s):
> The Header Parameter usage locations, which should be one or more of the values JWS or JWE.

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

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

9.1.2. Initial Registry Contents

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

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

* Header Parameter Name: `jku`
* Header Parameter Description: JWK Set URL
* Header Parameter Usage Location(s): JWS
* Change Controller: IESG

- Specification Document(s): **Section 4.1.2** of [[this document]]

- Header Parameter Name: `jwk`
- Header Parameter Description: JSON Web Key
- Header Parameter Usage Location(s): JWS
- Change Controller: IESG
- Specification document(s): **Section 4.1.3** of [[this document]]

- Header Parameter Name: `kid`
- Header Parameter Description: Key ID
- Header Parameter Usage Location(s): JWS
- Change Controller: IESG
- Specification Document(s): **Section 4.1.4** of [[this document]]

- Header Parameter Name: `x5u`
- Header Parameter Description: X.509 URL
- Header Parameter Usage Location(s): JWS
- Change Controller: IESG
- Specification Document(s): **Section 4.1.5** of [[this document]]

- Header Parameter Name: `x5c`
- Header Parameter Description: X.509 Certificate Chain
- Header Parameter Usage Location(s): JWS
- Change Controller: IESG
- Specification Document(s): **Section 4.1.6** of [[this document]]

- Header Parameter Name: `x5t`
- Header Parameter Description: X.509 Certificate SHA-1 Thumbprint
- Header Parameter Usage Location(s): JWS
- Change Controller: IESG
- Specification Document(s): **Section 4.1.7** of [[this document]]

- Header Parameter Name: `typ`
- Header Parameter Description: Type
- Header Parameter Usage Location(s): JWS
- Change Controller: IESG
- Specification Document(s): **Section 4.1.8** of [[this document]]

- Header Parameter Name: `cty`
- Header Parameter Description: Content Type
- Header Parameter Usage Location(s): JWS
- Change Controller: IESG
- Specification Document(s): **Section 4.1.9** of [[this document]]

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

9.2. Media Type Registration

9.2.1. Registry Contents

This specification registers the `application/jose` Media Type **[RFC2046]** in the MIME Media Types registry **[IANA.MediaTypes]**, 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.

- Type name: application

- Subtype name: jose
- Required parameters: n/a
- Optional parameters: n/a
- Encoding considerations: 8bit; application/jose values are encoded as a series of base64url encoded values (some of which may be the empty string) separated by period ('.') characters.
- Security considerations: See the Security Considerations section of [[this document]]
- Interoperability considerations: n/a
- Published specification: [[this document]]
- Applications that use this media type: OpenID Connect, Mozilla Persona, Salesforce, Google, Android, Windows Azure, Xbox One, and numerous others that use JWTs
- Additional information: Magic number(s): n/a, File extension(s): n/a, Macintosh file type code(s): n/a
- Person & email address to contact for further information: Michael B. Jones, mbj@microsoft.com
- Intended usage: COMMON
- Restrictions on usage: none
- Author: Michael B. Jones, mbj@microsoft.com
- Change Controller: IESG

- Type name: application
- Subtype name: jose+json
- Required parameters: n/a
- Optional parameters: n/a
- Encoding considerations: 8bit; application/jose+json values are represented as a JSON Object; UTF-8 encoding SHOULD be employed for the JSON object.
- Security considerations: See the Security Considerations section of [[this document]]
- Interoperability considerations: n/a
- Published specification: [[this document]]
- Applications that use this media type: TBD
- Additional information: Magic number(s): n/a, File extension(s): n/a, Macintosh file type code(s): n/a
- Person & email address to contact for further information: Michael B. Jones, mbj@microsoft.com
- Intended usage: COMMON
- Restrictions on usage: none
- Author: Michael B. Jones, mbj@microsoft.com
- Change Controller: IESG

10. Security Considerations

10.1. Cryptographic Security Considerations

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

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

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

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

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

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

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

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

10.2. JSON Security Considerations

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

Section 4 of the JSON Data Interchange Format specification **[I-D.ietf-json-rfc4627bis]** states "The names within an object SHOULD be unique", whereas this specification states that "Header Parameter names within this object MUST be unique; recipients MUST either reject JWSs with duplicate Header Parameter names or use a JSON parser that returns only the lexically last duplicate member name, as specified in Section 15.12 (The JSON Object) of ECMAScript 5.1 **[ECMAScript]**". Thus, this specification requires that the Section 4 "SHOULD" be treated as a "MUST" by senders and that it be either treated as a "MUST" or in the manner specified in ECMAScript 5.1 by receivers. 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 object followed by the extra characters ABCD. Such input MUST be rejected in its entirety.

10.3. Unicode Comparison Security Considerations

Header Parameter names and algorithm names are Unicode strings. For security reasons, the representations of these names must be compared verbatim after performing any escape processing (as per Section 8.3 of **[I-D.ietf-json-rfc4627bis]**). 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 (**HTML**, **PDF**).

[I-D.ietf-json-rfc4627bis]   Bray, T., "**The JSON Data Interchange Format**," draft-ietf-json-rfc4627bis-10 (work in progress), December 2013 (**TXT**).

[IANA.MediaTypes]   Internet Assigned Numbers Authority (IANA), "**MIME Media Types**," 2005.

[ITU.X690.1994]   International Telecommunications Union, "Information Technology - ASN.1 encoding rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER)," ITU-T Recommendation X.690, 1994.

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

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

[RFC1421]   **Linn, J.**, "**Privacy Enhancement for Internet Electronic Mail: Part I: Message Encryption and Authentication Procedures**," RFC 1421, February 1993 (**TXT**).

[RFC2045]   **Freed, N.** and **N. Borenstein**, "**Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies**," RFC 2045, November 1996 (**TXT**).

[RFC2046]   **Freed, N.** and **N. Borenstein**, "**Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types**," RFC 2046, November 1996 (**TXT**).

[RFC2119]   **Bradner, S.**, "**Key words for use in RFCs to Indicate Requirement Levels**," BCP 14, RFC 2119, March 1997 (**TXT**, **HTML**, **XML**).

[RFC2246]   **Dierks, T.** and **C. Allen**, "**The TLS Protocol Version 1.0**," RFC 2246, January 1999 (**TXT**).

[RFC2818]   Rescorla, E., "**HTTP Over TLS**," RFC 2818, May 2000 (**TXT**).

[RFC3629]   Yergeau, F., "**UTF-8, a transformation format of ISO 10646**," STD 63, RFC 3629, November 2003 (**TXT**).

[RFC3986]   **Berners-Lee, T.**, **Fielding, R.**, and **L. Masinter**, "**Uniform Resource Identifier (URI): Generic Syntax**," STD 66, RFC 3986, January 2005 (**TXT**, **HTML**, **XML**).

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

[RFC5226]   Narten, T. and H. Alvestrand, "**Guidelines for Writing an IANA Considerations Section in RFCs**," BCP 26, RFC 5226, May 2008 (**TXT**).

[RFC5246]   Dierks, T. and E. Rescorla, "**The Transport Layer Security (TLS) Protocol Version 1.2**," RFC 5246, August 2008 (**TXT**).

[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 (**TXT**).

[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 (**TXT**).

[USASCII]   American National Standards Institute, "Coded Character Set -- 7-bit American Standard Code for Information Interchange," ANSI X3.4, 1986.

[W3C.WD-xmldsig-bestpractices-20110809]   Datta, P. and F. Hirsch, "**XML Signature Best Practices**," World Wide Web Consortium WD WD-xmldsig-bestpractices-20110809, August 2011 (**HTML**).

11.2. Informative References

[CanvasApp]   Facebook, "**Canvas Applications**," 2010.

[JSS]   Bradley, J. and N. Sakimura (editor), "**JSON Simple Sign**," September 2010.

[JWE]   **Jones, M.**, **Rescorla, E.**, and **J. Hildebrand**, "**JSON Web Encryption (JWE)**," draft-ietf-jose-json-web-encryption (work in progress), January 2014 (**HTML**).

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

[MagicSignatures]   Panzer (editor), J., Laurie, B., and D. Balfanz, "**Magic Signatures**," January 2011.

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

[W3C.CR-xmldsig-core2-20120124]   Cantor, S., Roessler, T., Eastlake, D., Yiu, K., Reagle, J., Solo, D., Datta, P., and F. Hirsch, "**XML Signature Syntax and Processing Version 2.0**," World Wide Web Consortium CR CR-xmldsig-core2-20120124, January 2012 (**HTML**).

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"}
```

The octets representing UTF8(JWS Protected Header) in this case 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 of the JSON object above, is the JWS Payload:

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

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

```
eyJpc3MiOiJqb2UiLA0KICJleHAiOjEzMDA4MTkzODAsDQogImh0dHA6Ly9leGFt
cGxlLmNvbS9pc19yb290Ijp0cnVlfQ
```

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

```
eyJ0eXAiOiJKV1QiLA0KICJhbGciOiJIUzI1NiJ9
.
eyJpc3MiOiJqb2UiLA0KICJleHAiOjEzMDA4MTkzODAsDQogImh0dHA6Ly9leGFt
cGxlLmNvbS9pc19yb290Ijp0cnVlfQ
```

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

[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 for display purposes only):

```
{"kty":"oct",
 "k":"AyM1SysPpbyDfgZld3umj1qzKObwVMkoqQ-EstJQLr_T-1qS0gZH75
     aKtMN3Yj0iPS4hcgUuTwjAzZr1Z9CAow"
}
```

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

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

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

```
dBjftJeZ4CVP-mB92K27uhbUJU1p1r_wW1gFWFOEjXk
```

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

```
eyJ0eXAiOiJKV1QiLA0KICJhbGciOiJIUzI1NiJ9
.
eyJpc3MiOiJqb2UiLA0KICJleHAiOjEzMDA4MTkzODAsDQogImh0dHA6Ly9leGFt
cGxlLmNvbS9pc19yb290Ijp0cnVlfQ
.
dBjftJeZ4CVP-mB92K27uhbUJU1p1r_wW1gFWFOEjXk
```

A.1.2. Validating   [TOC]

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 as input to the HMAC SHA-256 function and then taking the output and determining if it matches the JWS Signature. If it matches exactly, the HMAC has been validated.

A.2. Example JWS using RSASSA-PKCS-v1_5 SHA-256   [TOC]

A.2.1. Encoding   [TOC]

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 case are:

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

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

```
eyJhbGciOiJSUzI1NiJ9
```

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

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

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

```
eyJhbGciOiJSUzI1NiJ9
.
eyJpc3MiOiJqb2UiLA0KICJleHAiOjEzMDA4MTkzODAsDQogImh0dHA6Ly9leGFt
cGxlLmNvbS9pc19yb290Ijp0cnVlfQ
```

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

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

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

```
{"kty":"RSA",
 "n":"ofgWCuLjybRlzo0tZWJjNiuSfb4p4fAkd_wWJcyQoTbji9k0l8W26mPddx
      HmfHQp-Vaw-4qPCJrcS2mJPMEzP1Pt0Bm4d4QlL-yRT-SFd2lZS-pCgNMs
      D1W_YpRPEwOWvG6b32690r2jZ47soMZo9wGzjb_7OMg0LOL-bSf63kpaSH
      SXndS5z5rexMdbBYUsLA9e-KXBdQOS-UTo7WTBEMa2R2CapHg665xsmtdV
      MTBQY4uDZlxvb3qCo5ZwKh9kG4LT6_I5IhlJH7aGhyxXFvUK-DWNmoudF8
      NAco9_h9iaGNj8q2ethFkMLs91kzk2PAcDTW9gb54h4FRWyuXpoQ",
 "e":"AQAB",
 "d":"Eq5xpGnNCivDflJsRQBXHx1hdR1k6Ulwe2JZD50LpXyWPEAeP88vLNO97I
      jlA7_GQ5sLKMgvfTeXZx9SE-7YwVol2NXOoAJe46sui395IW_GO-pWJ1O0
      BkTGoVEn2bKVRUCgu-GjBVaYLU6f3l9kJfFNS3E0QbVdxzubSu3Mkqzjkn
      439X0M_V51gfpRLI9JYanrC4D4qAdGcopV_0ZHHzQlBjudU2QvXt4ehNYT
      CBr6XCLQUShb1juUO1ZdiYoFaFQT5Tw8bGUl_x_jTj3ccPDVZFD9pIuhLh
      BOneufuBiB4cS98l2SR_RQyGWSeWjnczT0QU91p1DhOVRuOopznQ"
```

```
    }
```

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

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

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

```
cC4hiUPoj9Eetdgtv3hF80EGrhuB__dzERat0XF9g2VtQgr9PJbu3XOiZj5RZmh7
AAuHIm4Bh-0Qc_lF5YKt_O8W2Fp5jujGbds9uJdbF9CUAr7t1dnZcAcQjbKBYNX4
BAynRFdiuB--f_nZLgrnbyTyWzO75vRK5h6xBArLIARNPvkSjtQBMHlb1L07Qe7K
0GarZRmB_eSN9383LcOLn6_dO--xi12jzDwusC-eOkHWEsqtFZESc6BfI7noOPqv
hJ1phCnvWh6IeYI2w9QOYEUipUTI8np6LbgGY9Fs98rqVt5AXLIhWkWywlVmtVrB
p0igcN_IoypGlUPQGe77Rw
```

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

```
eyJhbGciOiJSUzI1NiJ9
.
eyJpc3MiOiJqb2UiLA0KICJleHAiOjEzMDA4MTkzODAsDQogImh0dHA6Ly9leGFt
cGxlLmNvbS9pc19yb290Ijp0cnVlfQ
.
cC4hiUPoj9Eetdgtv3hF80EGrhuB__dzERat0XF9g2VtQgr9PJbu3XOiZj5RZmh7
AAuHIm4Bh-0Qc_lF5YKt_O8W2Fp5jujGbds9uJdbF9CUAr7t1dnZcAcQjbKBYNX4
BAynRFdiuB--f_nZLgrnbyTyWzO75vRK5h6xBArLIARNPvkSjtQBMHlb1L07Qe7K
0GarZRmB_eSN9383LcOLn6_dO--xi12jzDwusC-eOkHWEsqtFZESc6BfI7noOPqv
hJ1phCnvWh6IeYI2w9QOYEUipUTI8np6LbgGY9Fs98rqVt5AXLIhWkWywlVmtVrB
p0igcN_IoypGlUPQGe77Rw
```

A.2.2. Validating

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

Validating the JWS Signature is a little different from the previous example. We pass (n, e), JWS Signature, and the JWS Signing Input 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 case are:

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

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

```
eyJhbGciOiJFUzI1NiJ9
```

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

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

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

```
eyJhbGciOiJFUzI1NiJ9
.
eyJpc3MiOiJqb2UiLA0KICJleHAiOjEzMDA4MTkzODAsDQogImh0dHA6Ly9leGFt
cGxlLmNvbS9pc19yb290Ijp0cnVlfQ
```

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

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

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

```
{"kty":"EC",
 "crv":"P-256",
 "x":"f83OJ3D2xF1Bg8vub9tLe1gHMzV76e8Tus9uPHvRVEU",
 "y":"x_FEzRu9m36HLN_tue659LNpXW6pCyStikYjKIWI5a0",
 "d":"jpsQnnGQmL-YBIffH1136cspYG6-0iY7X1fCE9-E9LI"
}
```

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

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

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

```
DtEhU3ljbEg8L38VWAfUAqOyKAM6-Xx-F4GawxaepmXFCgfTjDxw5djxLa8ISlSA
pmWQxfKTUJqPP3-Kg6NU1Q
```

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

```
eyJhbGciOiJFUzI1NiJ9
.
eyJpc3MiOiJqb2UiLA0KICJleHAiOjEzMDA4MTkzODAsDQogImh0dHA6Ly9leGFt
cGxlLmNvbS9pc19yb290Ijp0cnVlfQ
.
DtEhU3ljbEg8L38VWAfUAqOyKAM6-Xx-F4GawxaepmXFCgfTjDxw5djxLa8ISlSA
pmWQxfKTUJqPP3-Kg6NU1Q
```

A.3.2. Validating

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

Validating the JWS Signature is a little different from the first example. We need to split the 64 member octet sequence of the JWS Signature into two 32 octet sequences, the first R and the second S. We then pass (x, y), (R, S) and the JWS Signing Input 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 case 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 (with line breaks for display purposes only):

```
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 for display purposes only):

```
{"kty":"EC",
 "crv":"P-521",
 "x":"AekpBQ8ST8a8VcfVOTNl353vSrDCLLJXmPk06wTjxrrjcBpXp5EOnYG_
     NjFZ6OvLFV1jSfS9tsz4qUxcWceqwQGk",
 "y":"ADSmRA43Z1DSNx_RvcLI87cdL07l6jQyyBXMoxVg_l2Th-x3S1WDhjDl
     y79ajL4Kkd0AZMaZmh9ubmf63e3kyMj2",
 "d":"AY5pb7A0UFiB3RELSD64fTLOSV_jazdF7fLYyuTw8lOfRhWg6Y6rUrPA
     xerEzgdRhajnu0ferB0d53vM9mE15j2C"
}
```

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

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

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

```
AdwMgeerwtHoh-l192l60hp9wAHZFVJbLfD_UxMi70cwnZOYaRI1bKPWROc-mZZq
wqT2SI-KGDKB34X00aw_7XdtAG8GaSwFKdCAPZgoXD2YBJZCPEX3xKpRwcdOO8Kp
EHwJjyqOgzDO7iKvU8vcnwNrmxYbSW9ERBXukOXolLzeO_Jn
```

Concatenating these values in the order Header.Payload.Signature with period ('.') characters

between the parts yields this complete JWS representation using the JWS Compact Serialization (with line breaks for display purposes only):

```
eyJhbGciOiJFUzUxMiJ9

.

UGF5bG9hZA

.

AdwMgeerwtHoh-l192l60hp9wAHZFVJbLfD_UxMi70cwnZOYaRI1bKPWROc-mZZq
wqT2SI-KGDKB34XO0aw_7XdtAG8GaSwFKdCAPZgoXD2YBJZCPEX3xKpRwcdOO8Kp
EHwJjyqOgzDO7iKvU8vcnwNrmxYbSW9ERBXukOXolLzeO_Jn
```

A.4.2. Validating

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

Validating the JWS Signature is 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 R and the second S. We then pass (x, y), (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 Plaintext JWS

The following example JWS Protected Header declares that the encoded object is a Plaintext 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):

```
eyJhbGciOiJub25lIn0

.

eyJpc3MiOiJqb2UiLA0KICJleHAiOjEzMDA4MTkzODAsDQogImh0dHA6Ly9leGFt
cGxlLmNvbS9pc19yb290Ijp0cnVlfQ

.
```

A.6. Example JWS Using JWS JSON Serialization

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

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

```
eyJpc3MiOiJqb2UiLA0KICJleHAiOjEzMDA4MTkzODAsDQogImh0dHA6Ly9leGFt
cGxlLmNvbS9pc19yb290Ijp0cnVlfQ
```

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

A.6.1. JWS Per-Signature Protected Headers

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

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

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

```
eyJhbGciOiJSUzI1NiJ9
```

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

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

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

```
eyJhbGciOiJFUzI1NiJ9
```

A.6.2. JWS Per-Signature Unprotected Headers

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

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

and

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

A.6.3. Complete JWS Header Values

Combining the protected and unprotected header values supplied, the JWS 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 JSON Web Signature JSON Serialization for these values is as follows (with line breaks for display purposes only):

```
{"payload":
  "eyJpc3MiOiJqb2UiLA0KICJleHAiOjEzMDA4MTkzODAsDQogImh0dHA6Ly9leGF
   tcGxlLmNvbS9pc19yb290Ijp0cnVlfQ",
 "signatures":[
  {"protected":"eyJhbGciOiJSUzI1NiJ9",
   "header":
    {"kid":"2010-12-29"},
   "signature":
    "cC4hiUPoj9Eetdgtv3hF80EGrhuB__dzERat0XF9g2VtQgr9PJbu3XOiZj5RZ
     mh7AAuHIm4Bh-0Qc_lF5YKt_O8W2Fp5jujGbds9uJdbF9CUAr7t1dnZcAcQjb
     KBYNX4BAynRFdiuB--f_nZLgrnbyTyWzO75vRK5h6xBArLIARNPvkSjtQBMHl
     b1L07Qe7K0GarZRmB_eSN9383LcOLn6_dO--xi12jzDwusC-eOkHWEsqtFZES
     c6BfI7noOPqvhJ1phCnvWh6IeYI2w9QOYEUipUTI8np6LbgGY9Fs98rqVt5AX
     LIhWkWywlVmtVrBp0igcN_IoypGlUPQGe77Rw"},
  {"protected":"eyJhbGciOiJFUzI1NiJ9",
   "header":
    {"kid":"e9bc097a-ce51-4036-9562-d2ade882db0d"},
   "signature":
    "DtEhU3ljbEg8L38VWAfUAqOyKAM6-Xx-F4GawxaepmXFCgfTjDxw5djxLa8IS
     lSApmWQxfKTUJqPP3-Kg6NU1Q"}]
}
```

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.

```
["MIIE3jCCA8agAwIBAgICAwEwDQYJKoZIhvcNAQEFBQAwYzELMAkGA1UEBhMCVVM
  xITAfBgNVBAoTGFRoZSBHbyBEYWRkeSBHcm91cCwgSW5jLjExMC8GA1UECxMoR2
  8gRGFkZHkgQ2xhc3MgMiBDZXJ0aWZpY2F0aW9uIEF1dGhvcml0eTAeFw0wNjExM
  TYwMTU0MzdaFw0yNjExMTYwMTU0MzdaMIHKMQswCQYDVQQGEwJVUzEQMA4GA1UE
  CBMHQXJpem9uYTETMBEGA1UEBxMKU2NvdHRzZGFsZTEaMBgGA1UEChMRR29EYWR
  keS5jb20sIEluYy4xMzAxBgNVBAsTKmh0dHA6Ly9jZXJ0aWZpY2F0ZXMuZ29kYW
```

RkeS5jb20vcmVwb3NpdG9yeTEwMC4GA1UEAxMnR28gRGFkZHkgU2VjdXJlIENlc
nRpZmljYXRpb24gQXV0aG9yaXR5MREwDwYDVQQFEwgwNzk2OTI4NzCCASIwDQYJ
KoZIhvcNAQEBBQADggEPADCCAQoCggEBAMQt1RWMnCZM7DI161+4WQFapmGBWTt
wY6vj3D3HKrjJM9N55DrtPDAjhI6zMBS2sofDPZVUBJ7fmd0LJR4h3mUpfjWoqV
Tr9vcyOdQmVZWt7/v+WIbXnvQAjYwqDL1CBM6nPwT27oDyqu9SoWlm2r4arV3aL
GbqGmu75RpRSgAvSMeYddi5Kcju+GZtCpyz8/x4fKL4o/K1w/O5epHBp+YlLpyo
7RJlbmr2EkRTcDCVw5wrWCs9CHRK8r5RsL+H0EwnWGu1NcWdrxcx+AuP7q2BNgW
JCJjPOq8lh8BJ6qf9Z/dFjpfMFDniNoW1fho3/Rb2cRGadDAW/hOUoz+EDU8CAw
EAAaOCATIwggEuMB0GA1UdDgQWBBT9rGEyk2xF1uLuhV+auud2mWjM5zAfBgNVH
SMEGDAWgBTSxLDSkdRMEXGzYcs9of7dqGrU4zASBgNVHRMBAf8ECDAGAQH/AgEA
MDMGCCsGAQUFBwEBBCcwJTAjBggrBgEFBQcwAYYXaHR0cDovL29jc3AuZ29kYWR
keS5jb20wRgYDVR0fBD8wPTA7oDmgN4Y1aHR0cDovL2NlcnRpZmljYXRlcy5nb2
RhZGR5LmNvbS9yZXBvc2l0b3J5L2dkcm9vdC5jcmwwSwYDVR0gBEQwQjBABgRVH
SAAMDgwNgYIKwYBBQUHAgEWKmh0dHA6Ly9jZXJ0aWZpY2F0ZXMuZ29kYWRkeS5j
b20vcmVwb3NpdG9yeTAOBgNVHQ8BAf8EBAMCAYwDQYJKoZIhvcNAQEFBQADggE
BANKGwOy9+aG2Z+5mC6IGOgRQjhVyrEp0lVPLN8tESe8HkGsz2ZbwlFalEzAFPI
UyIXvJxwqoJKSQ3kbTJSMUA2fCENZvD117esyfxVgqwcSeIaha86ykRvOe5GPLL
5CkKSkB2XIsKd83ASe8T+5o0yGPwLPk9Qnt0hCqU7S+8MxZC9Y7lhyVJEnfzuz9
p0iRFEUOOjZv2kWzRaJBydTXRE4+uXR21aITVSzGh6O1mawGhId/dQb8vxRMDsx
uxN89txJx9OjxUUAiKEngHUuHqDTMBqLdElrRhjZkAzVvb3du6/KFUJheqwNTrZ
EjYx8WnM25sgVjOuH0aBsXBTWVU+4=",
"MIIE+zCCBGSgAwIBAgICAQ0wDQYJKoZIhvcNAQEFBQAwgbsxJDAiBgNVBAcTG1Z
hbGlDZXJ0IFZhbGlkYXRpb24gTmV0d29yazEXMBUGA1UEChMOVmFsaUNlcnQsIE
luYy4xNTAzBgNVBAsTLFZhbGlDZXJ0IENsYXNzIDIgUG9saWN5IFZhbGlkYXRpb
24gQXV0aG9yaXR5MSEwHwYDVQQDExhodHRwOi8vd3d3LnZhbGljZXJ0LmNvbS8x
IDAeBgkqhkiG9w0BCQEWEWluZm9AdmFsaWNlcnQuY29tMB4XDTA0MDYyOTE3MDY
yMFoXDTI0MDYyOTE3MDYyMFowYzELMAkGA1UEBhMCVVMxITAfBgNVBAoTGFRoZS
BHbyBEYWRkeSBHcm91cCwgSW5jLjExMC8GA1UECxMoR28gRGFkZHkgQ2xhc3MgM
iBDZXJ0aWZpY2F0aW9uIEF1dGhvcml0eTCCASAwDQYJKoZIhvcNAQEBBQADggEN
ADCCAQgCggEBAN6d1+pXGEmhW+vXX0iG6r7d/+TvZxz0ZWizV3GgXne77ZtJ6XC
APVYYYwhv2vLM0D9/AlQiVBDYsoHUwHU9S3/Hd8M+eKsaA7Ugay9qK7HFiH7Eux
6wwdhFJ2+qN1j3hybX2C32qRe3H3I2TqYXP2WYktsqbl2i/ojgC95/5Y0V4evLO
tXiEqITLdiOr18SPaAIBQi2XKVlOARFmR6jYGB0xUGlcmIbYsUfb18aQr4CUWWo
riMYavx4A6lNf4DD+qta/KFApMoZFv6yyO9ecw3ud72a9nmYvLEHZ6IVDd2gWMZ
Eewo+YihfukEHU1jPEX44dMX4/7VpkI+EdOqXG68CAQOjggHhMIIB3TAdBgNVHQ
4EFgQU0sSw0pHUTBFxs2HLPaH+3ahq1OMwgdIGA1UdIwSByjCBx6GBwaSBvjCBu
zEkMCIGA1UEBxMbVmFsaUNlcnQgVmFsaWRhdGlvbiBOZXR3b3JrMRcwFQYDVQQK
Ew5WYWxpQ2VydCwgSW5jLjE1MDMGA1UECxMsVmFsaUNlcnQgQ2xhc3MgMiBQb2x
pY3kgVmFsaWRhdGlvbiBBdXRob3JpdHkxITAfBgNVBAMTGGh0dHA6Ly93d3cudm
FsaWNlcnQuY29tLzEgMB4GCSqGSIb3DQEJARYRaW5mb0B2YWxpY2VydC5jb22CA
QEwDwYDVR0TAQH/BAUwAwEB/zAzBggrBgEFBQcBAQQnMCUwIwYIKwYBBQUHMAGG
F2h0dHA6Ly9vY3NwLmdvZGFkZHkuY29tMEQGA1UdHwQ9MDswOaA3oDWGM2h0dHA
6Ly9jZXJ0aWZpY2F0ZXMuZ29kYWRkeS5jb20vcmVwb3NpdG9yeS9yb290LmNybD
BLBgNVHSAERDBCMEAGBFUdIAAwODA2BggrBgEFBQcCARYqaHR0cDovL2NlcnRpZ
mljYXRlcy5nb2RhZGR5LmNvbS9yZXBvc2l0b3J5MA4GA1UdDwEB/wQEAwIBBjAN
BgkqhkiG9w0BAQUFAAOBgQC1QPmnHfbq/qQaQlpE9xXUhUaJwL6e4+PrxeNYiY+
Sn1eocSxI0YGyeR+sBjUZsE4OWBsUs5iB0QQeyAfJg594RAoYC5jcdnplDQ1tgM
QLARzLrUc+cb53S8wGd9D0VmsfSxOaFIqII6hR8INMqzW/Rn453HWkrugp++85j
09VZw==",
"MIIC5zCCAlACAQEwDQYJKoZIhvcNAQEFBQAwgbsxJDAiBgNVBAcTG1ZhbGlDZXJ
0IFZhbGlkYXRpb24gTmV0d29yazEXMBUGA1UEChMOVmFsaUNlcnQsIEluYy4xNT
AzBgNVBAsTLFZhbGlDZXJ0IENsYXNzIDIgUG9saWN5IFZhbGlkYXRpb24gQXV0a
G9yaXR5MSEwHwYDVQQDExhodHRwOi8vd3d3LnZhbGljZXJ0LmNvbS8xIDAeBgkq
hkiG9w0BCQEWEWluZm9AdmFsaWNlcnQuY29tMB4XDTk5MDYyNjAwMTk1NFoXDTE
5MDYyNjAwMTk1FowgbsxJDAiBgNVBAcTG1ZhbGlDZXJ0IFZhbGlkYXRpb24gTm
V0d29yazEXMBUGA1UEChMOVmFsaUNlcnQsIEluYy4xNTAzBgNVBAsTLFZhbGlDZ
XJ0IENsYXNzIDIgUG9saWN5IFZhbGlkYXRpb24gQXV0aG9yaXR5MSEwHwYDVQQD
ExhodHRwOi8vd3d3LnZhbGljZXJ0LmNvbS8xIDAeBgkqhkiG9w0BCQEWEWluZm9
AdmFsaWNlcnQuY29tMIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQDOnHK5a
vIWZJV16vYdA757tn2VUdZZUcOBVXc65g2PFxTXdMwzzjsvUGJ7SVCCSRrCl6zf
N1SLUzm1NZ9WlmpZdRJEy0kTRxQb7XBhVQ7/nHk01xC+YDgkRoKWzk2Z/M/VXwb
P7RfZHM047QSv4dk+NoS/zcnwbNDu+97bi5p9wIDAQABMA0GCSqGSIb3DQEBBQU
AA4GBADt/UG9vUJSZSWI4OB9L+KXIPqeCgfYrx+jFzug6EILLGACOTb2oWH+heQ
C1u+mNr0HZDzTuIYEZoDJJKPTEjlbVUjP9UNV+mWwD5lM/Mtsq2azSiGM5bUMM
j4QssxsodyamEwCW/POuZ6lcg5Ktz885hZo+L7tdEy8W9ViH0Pd"]

Appendix C. Notes on implementing base64url encoding without padding

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

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

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

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

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

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

```
3 236 255 224 193
```

```
A-z_4ME
```

Appendix D. Notes on Key Selection

This appendix describes a set of possible algorithms for selecting the key to be used to validate the digital signature or MAC of a JWS 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.

These algorithms use some or all of the steps described below; the order and inclusion of the steps does not mean that they need to be performed in this order or that they are all

required in all contexts. The steps below are described for illustration purposes only; specific applications can and are likely to use different algorithms. 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**.

Algorithm steps may include:

1. Collect a 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.
   - Keys 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.
2. 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 other keys. Likewise, keys with certain `kty` (Key Type) values, `alg` (Algorithm) values, or other member values might be ordered before keys with other `kty` values, `alg` values, or other member values.
3. Filter the set of collected keys. For instance, only keys referenced by `kid` (key ID) or `x5t` (X.509 certificate SHA-1 thumbprint) parameters might be tried by the application. Keys with an invalid certificate chain would typically be excluded. Keys with inappropriate `alg` (algorithm), `use` (public key use), or `key_ops` (key operations) values would likewise typically be excluded. Keys might be filtered to include or exclude keys with certain `kty` (key type) values or other member values. A limit on the number of keys to be tried might also be applied.
4. Attempt signature or MAC validation for a JWS object or decryption of a JWE object with some or all of the collected and possibly ordered and/or filtered keys. This process will normally terminate following a successful validation or decryption.

Appendix E. Negative Test Case for "crit" Header Parameter

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

The JWS Protected Header value for this JWS is:

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

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

```
eyJhbGciOiJub25lIiwNCiAiY3JpdCI6WyJodHRwOi8vZXhhbXBsZS5jb20vVU5ERU
ZJTkVEIl0sDQogImh0dHA6Ly9leGFtcGxlLmNvbS9VTkRFRklORUQiOnRydWUNCn0.
RkFJTA.
```

Appendix F. Detached Content

In some contexts, it is useful integrity protect content that is not itself contained in a JWS 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, Breno de Medeiros, Dick Hardt, Joe Hildebrand, Jeff Hodges, Edmund Jay, Yaron Y. Goland, Ben Laurie, James Manger, Matt Miller, Tony Nadalin, Axel Nennker, John Panzer, Emmanuel Raviart, Eric Rescorla, Jim Schaad, Paul Tarjan, Hannes Tschofenig, and Sean Turner.

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

Appendix H. Document History

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

-20

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

-19

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

-18

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

- Added and used Header Parameter Description registry field.

-17

- Refined the `typ` and `cty` definitions to always be MIME Media Types, with the omission of "application/" prefixes recommended for brevity, addressing issue #50.
- Updated the mandatory-to-implement (MTI) language to say that general-purpose implementations must implement the single signature/MAC value case for both serializations whereas special-purpose implementations can implement just one serialization if that meets the needs of the use cases the implementation is designed for, addressing issue #119.
- Explicitly named all the logical components of a JWS and defined the processing rules and serializations in terms of those components, addressing issues #60, #61, and #62.
- Replaced verbose repetitive phases such as "base64url encode the octets of the UTF-8 representation of X" with mathematical notation such as "BASE64URL(UTF8(X))".
- Terms used in multiple documents are now defined in one place and incorporated by reference. Some lightly used or obvious terms were also removed. This addresses issue #58.

-16

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

-15

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

-14

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

-13

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

-12

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

-11

- Added Key Identification section.
- For the JWS JSON Serialization, enable header parameter values to be specified in any of three parameters: the `protected` member that is integrity protected and shared among all recipients, the `unprotected` member that is not integrity protected and shared among all recipients, and the `header` member that is not integrity protected and specific to a particular recipient. (This does not affect the JWS Compact Serialization, in which all header parameter values are in a single

integrity protected JWE Header value.)

- Removed suggested compact serialization for multiple digital signatures and/or MACs.
- Changed the MIME type name `application/jws-js` to `application/jws+json`, addressing issue #22.
- Tightened the description of the `crit` (critical) header parameter.
- Added a negative test case for the `crit` header parameter

-10

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

-09

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

-08

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

-07

- Updated references.

-06

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

-05

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

-04

- Completed JSON Security Considerations section, including considerations about rejecting input with duplicate member names.
- Completed security considerations on the use of a SHA-1 hash when computing `x5t` (x.509 certificate thumbprint) values.
- Refer to the registries as the primary sources of defined values and then secondarily reference the sections defining the initial contents of the registries.
- Normatively reference **XML DSIG 2.0** [W3C.CR-xmldsig-core2-20120124] for its security considerations.
- Added this language to Registration Templates: "This name is case sensitive.

Names that match other registered names in a case insensitive manner SHOULD NOT be accepted."
- Reference draft-jones-jose-jws-json-serialization instead of draft-jones-json-web-signature-json-serialization.
- Described additional open issues.
- Applied editorial suggestions.

-03

- Added the `cty` (content type) header parameter for declaring type information about the secured content, as opposed to the `typ` (type) header parameter, which declares type information about this object.
- Added "Collision Resistant Namespace" to the terminology section.
- Reference ITU.X690.1994 for DER encoding.
- Added an example JWS using ECDSA P-521 SHA-512. This has particular illustrative value because of the use of the 521 bit integers in the key and signature values. This is also an example in which the payload is not a base64url encoded JSON object.
- Added an example `x5c` value.
- No longer say "the UTF-8 representation of the JWS Secured Input (which is the same as the ASCII representation)". Just call it "the ASCII representation of the JWS Secured Input".
- Added Registration Template sections for defined registries.
- Added Registry Contents sections to populate registry values.
- Changed name of the JSON Web Signature and Encryption "typ" Values registry to be the JSON Web Signature and Encryption Type Values registry, since it is used for more than just values of the `typ` parameter.
- Moved registries JSON Web Signature and Encryption Header Parameters and JSON Web Signature and Encryption Type Values to the JWS specification.
- Numerous editorial improvements.

-02

- Clarified that it is an error when a `kid` value is included and no matching key is found.
- Removed assumption that `kid` (key ID) can only refer to an asymmetric key.
- Clarified that JWSs with duplicate Header Parameter Names MUST be rejected.
- Clarified the relationship between `typ` header parameter values and MIME types.
- Registered application/jws MIME type and "JWS" typ header parameter value.
- Simplified JWK terminology to get replace the "JWK Key Object" and "JWK Container Object" terms with simply "JSON Web Key (JWK)" and "JSON Web Key Set (JWK Set)" and to eliminate potential confusion between single keys and sets of keys. As part of this change, the Header Parameter Name for a public key value was changed from `jpk` (JSON Public Key) to `jwk` (JSON Web Key).
- Added suggestion on defining additional header parameters such as `x5t#S256` in the future for certificate thumbprints using hash algorithms other than SHA-1.
- Specify RFC 2818 server identity validation, rather than RFC 6125 (paralleling the same decision in the OAuth specs).
- Generalized language to refer to Message Authentication Codes (MACs) rather than Hash-based Message Authentication Codes (HMACs) unless in a context specific to HMAC algorithms.
- Reformatted to give each header parameter its own section heading.

-01

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

- Changed "no canonicalization is performed" to "no canonicalization need be performed".
- Corrected the Magic Signatures reference.
- Made other editorial improvements suggested by JOSE working group participants.

-00

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

Authors' Addresses

Michael B. Jones
Microsoft
Email: **mbj@microsoft.com**
URI: **http://self-issued.info/**

John Bradley
Ping Identity
Email: **ve7jtb@ve7jtb.com**
URI: **http://www.thread-safe.com/**

Nat Sakimura
Nomura Research Institute
Email: **n-sakimura@nri.co.jp**
URI: **http://nat.sakimura.org/**