JOSE Working Group	M. Jones
Internet-Draft	Microsoft
Intended status: Standards Track	J. Bradley
Expires: January 17, 2013	Ping Identity
	N. Sakimura
	NRI
	July 16, 2012

TOC

JSON Web Signature (JWS)
draft-ietf-jose-json-web-signature-04

Abstract

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

Status of this Memo

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

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

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

This Internet-Draft will expire on January 17, 2013.

Table of Contents

---

1. Introduction

TOC

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

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

TOC

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]**.

TOC

2. Terminology

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

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

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

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

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

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

Encoded JWS Payload
: Base64url encoding of the JWS Payload.

Encoded JWS Signature
: Base64url encoding of the JWS Signature.

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

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

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

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

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

StringOrURI
: A JSON string value, with the additional requirement that while arbitrary string values MAY be used, any value containing a ":" character MUST be a URI **[RFC3986]**.

TOC

3. JSON Web Signature (JWS) Overview

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

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

TOC

3.1. Example JWS

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

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

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

```
eyJ0eXAiOiJKV1QiLA0KICJhbGciOiJIUzI1NiJ9
```

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

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

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

```
eyJpc3MiOiJqb2UiLA0KICJleHAiOjEzMDA4MTkzODAsDQogImh0dHA6Ly9leGFt
cGxlLmNvbS9pc19yb290Ijp0cnVlfQ
```

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

```
dBjftJeZ4CVP-mB92K27uhbUJU1p1r_wW1gFWFOEjXk
```

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

```
eyJ0eXAiOiJKV1QiLA0KICJhbGciOiJIUzI1NiJ9
.
eyJpc3MiOiJqb2UiLA0KICJleHAiOjEzMDA4MTkzODAsDQogImh0dHA6Ly9leGFt
cGxlLmNvbS9pc19yb290Ijp0cnVlfQ
```

```
.
dBjftJeZ4CVP-mB92K27uhbUJU1p1r_wW1gFWFOEjXk
```

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

TOC

4. JWS Header

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

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

TOC

4.1. Reserved Header Parameter Names

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

Additional reserved header parameter names MAY be defined via the IANA JSON Web Signature and Encryption Header Parameters registry **Section 7.1**. As indicated by the common registry, JWSs and JWEs share a common header parameter space; when a parameter is used by both specifications, its usage must be compatible between the specifications.

TOC

4.1.1. "alg" (Algorithm) Header Parameter

The `alg` (algorithm) header parameter identifies the cryptographic algorithm used to secure the JWS. The algorithm specified by the `alg` value MUST be supported by the implementation and there MUST be a key for use with that algorithm associated with the party that digitally signed or MACed the content or the JWS MUST be rejected. `alg` values SHOULD either be registered in the IANA JSON Web Signature and Encryption Algorithms registry **[JWA]** or be a URI that contains a Collision Resistant Namespace. The `alg` value is a case sensitive string containing a StringOrURI value. This header parameter is REQUIRED.

A list of defined `alg` values can be found in the IANA JSON Web Signature and Encryption Algorithms registry **[JWA]**; the initial contents of this registry is the values defined in Section 3.1 of the JSON Web Algorithms (JWA) **[JWA]** specification.

TOC

4.1.2. "jku" (JWK Set URL) Header Parameter

The `jku` (JWK Set URL) header parameter is a URI **[RFC3986]** that refers to a resource for a set of JSON-encoded public keys, one of which corresponds to the key used to digitally sign the JWS. The keys MUST be encoded as a JSON Web Key Set (JWK Set) **[JWK]**. The protocol used to acquire the resource MUST provide integrity protection; an HTTP GET request to retrieve the certificate MUST use TLS **[RFC2818]** **[RFC5246]**; the identity of the server MUST be validated, as per Section 3.1 of HTTP Over TLS **[RFC2818]**. This header parameter is OPTIONAL.

TOC

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

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

TOC

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

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

TOC

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

The x5t (X.509 Certificate Thumbprint) header parameter provides a base64url encoded SHA-1 thumbprint (a.k.a. digest) of the DER encoding of the X.509 certificate **[RFC5280]** corresponding to the key used to digitally sign the JWS. This header parameter is OPTIONAL.

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

TOC

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

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

See **Appendix B** for an example x5c value.

TOC

4.1.7. "kid" (Key ID) Header Parameter

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

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

TOC

4.1.8. "typ" (Type) Header Parameter

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

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

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

TOC

4.1.9. "cty" (Content Type) Header Parameter

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

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

TOC

4.2. Public Header Parameter Names

Additional header parameter names can be defined by those using JWSs. However, in order to prevent collisions, any new header parameter name SHOULD either be registered in the IANA JSON Web Signature and Encryption Header Parameters registry **Section 7.1** or be a URI that contains a Collision Resistant Namespace. In each case, the definer of the name or value needs to take reasonable precautions to make sure they are in control of the part of the namespace they use to define the header parameter name.

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

TOC

4.3. Private Header Parameter Names

A producer and consumer of a JWS may agree to any header parameter name that is not a Reserved Name **Section 4.1** or a Public Name **Section 4.2**. Unlike Public Names, these private names are subject to collision and should be used with caution.

TOC

5. Rules for Creating and Validating a JWS

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

1. Create the content to be used as the JWS Payload.
2. Base64url encode the bytes of the JWS Payload. This encoding becomes the Encoded JWS Payload.
3. Create a JWS Header containing the desired set of header parameters. Note that white space is explicitly allowed in the representation and no canonicalization need be performed before encoding.
4. Base64url encode the bytes of the UTF-8 representation of the JWS Header to create the Encoded JWS Header.
5. Compute the JWS Signature in the manner defined for the particular algorithm being used. The JWS Secured Input is always the concatenation of the Encoded JWS Header, a period ('.') character, and the Encoded JWS Payload. The `alg` (algorithm) header parameter MUST be present in the JSON Header, with the algorithm value accurately representing the algorithm used to construct the JWS Signature.

6. Base64url encode the representation of the JWS Signature to create the Encoded JWS Signature.
7. The three encoded parts, taken together, are the result. The Compact Serialization of this result is the concatenation of the Encoded JWS Header, the Encoded JWS Payload, and the Encoded JWS Signature in that order, with the three strings being separated by period ('.') characters.

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

1. Parse the three parts of the input (which are separated by period characters when using the JWS Compact Serialization) into the Encoded JWS Header, the Encoded JWS Payload, and the Encoded JWS Signature.
2. The Encoded JWS Header MUST be successfully base64url decoded following the restriction given in this specification that no padding characters have been used.
3. The resulting JWS Header MUST be completely valid JSON syntax conforming to **RFC 4627** [RFC4627].
4. The resulting JWS Header MUST be validated to only include parameters and values whose syntax and semantics are both understood and supported.
5. The Encoded JWS Payload MUST be successfully base64url decoded following the restriction given in this specification that no padding characters have been used.
6. The Encoded JWS Signature MUST be successfully base64url decoded following the restriction given in this specification that no padding characters have been used.
7. The JWS Signature MUST be successfully validated against the JWS Secured Input (the concatenation of the Encoded JWS Header, a period ('.') character, and the Encoded JWS Payload) in the manner defined for the algorithm being used, which MUST be accurately represented by the value of the `alg` (algorithm) header parameter, which MUST be present.

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

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

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

---

6. Securing JWSs with Cryptographic Algorithms

TOC

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

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

---

7. IANA Considerations

TOC

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

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

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

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

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

TOC

7.1. JSON Web Signature and Encryption Header Parameters Registry

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

TOC

7.1.1. Registration Template

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

Change Controller:
: For standards-track RFCs, state "IETF". For others, give the name of the responsible party. Other details (e.g., postal address, e-mail address, home page URI) may also be included.

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

TOC

7.1.2. Initial Registry Contents

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

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

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

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

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

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

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

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

- Header Parameter Name: `typ`
- Change Controller: IETF
- Specification Document(s): **Section 4.1.8** of [[this document]]

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

TOC

7.2. JSON Web Signature and Encryption Type Values Registry

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

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

TOC

7.2.1. Registration Template

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

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

Change Controller:
: For standards-track RFCs, state "IETF". For others, give the name of the responsible party. Other details (e.g., postal address, e-mail address, home page URI) may also be included.

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

TOC

7.2.2. Initial Registry Contents

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

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

7.3. Media Type Registration

TOC

7.3.1. Registry Contents

TOC

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

- Type name: application
- Subtype name: jws
- Required parameters: n/a
- Optional parameters: n/a
- Encoding considerations: JWS values are encoded as a series of base64url encoded values (some of which may be the empty string) separated by period ('.') characters
- 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 Browser ID, Salesforce, Google, numerous others that use signed 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: IETF

---

8. Security Considerations

TOC

8.1. Cryptographic Security Considerations

TOC

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

All the security considerations in **XML DSIG 2.0** [W3C.CR-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.

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.

TOC

8.2. JSON Security Considerations

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

Section 2.2 of the JavaScript Object Notation (JSON) specification **[RFC4627]** states "The names within an object SHOULD be unique", whereas this specification states that "Header Parameter Names within this object MUST be unique; JWSs with duplicate Header Parameter Names MUST be rejected". Thus, this specification requires that the Section 2.2 "SHOULD" be treated as a "MUST". Ambiguous and potentially exploitable situations could arise if the JSON parser used does not enforce the uniqueness of member names.

TOC

8.3. Unicode Comparison Security Considerations

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

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

TOC

9. Open Issues

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

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

- Should we define an optional nonce and/or timestamp header parameter? (Use of a nonce is an effective countermeasure to some kinds of attacks.)
- Some have objected to the language "Implementations MUST understand the entire contents of the header; otherwise, the JWS MUST be rejected" in this spec and the related language in the JWE, JWK, and JWT specs. Others believe that this is essential in a security specification.
- Finish the Security Considerations section.

TOC

10. References

10.1. Normative References

TOC

[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)," July 2012.

[JWK] Jones, M., "JSON Web Key (JWK)," July 2012.

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

[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).

[RFC4288] Freed, N. and J. Klensin, "Media Type Specifications and Registration Procedures," BCP 13, RFC 4288, December 2005 (TXT).

[RFC4627] Crockford, D., "The application/json Media Type for JavaScript Object Notation (JSON)," RFC 4627, July 2006 (TXT).

[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).

[USA15] Davis, M., Whistler, K., and M. Dürst, "Unicode Normalization Forms," Unicode Standard Annex 15, 09 2009.

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

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

10.2. Informative References

TOC

[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)," July 2012.

[JWS-JS] Jones, M., Bradley, J., and N. Sakimura, "JSON Web Signature JSON Serialization (JWS-JS)," July 2012.

[JWT] Jones, M., Bradley, J., and N. Sakimura, "JSON Web Token (JWT)," July 2012.

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

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

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

Appendix A. JWS Examples

TOC

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

A.1. JWS using HMAC SHA-256

TOC

A.1.1. Encoding

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

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

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

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

Base64url encoding these bytes yields this Encoded JWS Header value:

```
eyJ0eXAiOiJKV1QiLA0KICJhbGciOiJIUzI1NiJ9
```

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

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

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

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

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

```
eyJpc3MiOiJqb2UiLA0KICJleHAiOjEzMDA4MTkzODAsDQogImh0dHA6Ly9leGFt
cGxlLmNvbS9pc19yb290Ijp0cnVlfQ
```

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

```
eyJ0eXAiOiJKV1QiLA0KICJhbGciOiJIUzI1NiJ9
.
eyJpc3MiOiJqb2UiLA0KICJleHAiOjEzMDA4MTkzODAsDQogImh0dHA6Ly9leGFt
cGxlLmNvbS9pc19yb290Ijp0cnVlfQ
```

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

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

HMACs are generated using keys. This example uses the key represented by the following byte array:

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

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

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

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

```
dBjftJeZ4CVP-mB92K27uhbUJU1p1r_wW1gFWFOEjXk
```

TOC

A.1.2. Decoding

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

TOC

A.1.3. Validating

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

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

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

TOC

A.2. JWS using RSA SHA-256

TOC

A.2.1. Encoding

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

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

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

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

Base64url encoding these bytes yields this Encoded JWS Header value:

```
eyJhbGciOiJSUzI1NiJ9
```

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

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

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

```
eyJhbGciOiJSUzI1NiJ9
.
eyJpc3MiOiJqb2UiLA0KICJleHAiOjEzMDA4MTkzODAsDQogImh0dHA6Ly9leGFt
cGxlLmNvbS9pc19yb290Ijp0cnVlfQ
```

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

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

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

Parameter Name	Value
Modulus	[161, 248, 22, 10, 226, 227, 201, 180, 101, 206, 141, 45, 101, 98, 99, 54, 43, 146, 125, 190, 41, 225, 240, 36, 119, 252, 22, 37, 204, 144, 161, 54, 227, 139, 217, 52, 151, 197, 182, 234, 99, 221, 119, 17, 230, 124, 116, 41, 249, 86, 176, 251, 138, 143, 8, 154, 220, 75, 105, 137, 60, 193, 51, 63, 83, 237, 208, 25, 184, 119, 132, 37, 47, 236, 145, 79, 228, 133, 119, 105, 89, 75, 234, 66, 128, 211, 44, 15, 85, 191, 98, 148, 79, 19, 3, 150, 188, 110, 155, 223, 110, 189, 210, 189, 163, 103, 142, 236, 160, 198, 104, 247, 1, 179, 141, 191, 251, 56, 200, 52, 44, 226, 254, 109, 39, 250, 222, 74, 90, 72, 116, 151, 157, 212, 185, 207, 154, 222, 196, 199, 91, 5, 133, 44, 44, 15, 94, 248, 165, 193, 117, 3, 146, 249, 68, 232, 237, 100, 193, 16, 198, 182, 71, 96, 154, 164, 120, 58, 235, 156, 108, 154, 215, 85, 49, 48, 80, 99, 139, 131, 102, 92, 111, 111, 122, 130, 163, 150, 112, 42, 31, 100, 27, 130, 211, 235, 242, 57, 34, 25, 73, 31, 182, 134, 135, 44, 87, 22, 245, 10, 248, 53, 141, 154, 139, 157, 23, 195, 64, 114, 143, 127, 135, 216, 154, 24, 216, 252, 171, 103, 173, 132, 89, 12, 46, 207, 117, 147, 57, 54, 60, 7, 3, 77, 111, 96, 111, 158, 33, 224, 84, 86, 202, 229, 233, 161]
Exponent	[1, 0, 1]
Private Exponent	[18, 174, 113, 164, 105, 205, 10, 43, 195, 126, 82, 108, 69, 0, 87, 31, 29, 97, 117, 29, 100, 233, 73, 112, 123, 98, 89, 15, 157, 11, 165, 124, 150, 60, 64, 30, 63, 207, 47, 44, 211, 189, 236, 136, 229, 3, 191, 198, 67, 155, 11, 40, 200, 47, 125, 55, 151, 103, 31, 82, 19, 238, 216, 193, 90, 37, 216, 213, 206, 160, 2, 94, 227, 171, 46, 139, 127, 121, 33, 111, 198, 59, 234, 86, 39, 83, 180, 6, 68, 198, 161, 81, 39, 217, 178, 149, 69, 64, 160, 187, 225, 163, 5, 86, 152, 45, 78, 159, 222, 95, 100, 37, 241, 77, 75, 113, 52, 65, 181, 93, 199, 59, 155, 74, 237, 204, 146, 172, 227, 146, 126, 55, 245, 125, 12, 253, 94, 117, 129, 250, 81, 44, 143, 73, 97, 169, 235, 11, 128, 248, 168, 7, 70, 114, 138, 85, 255, 70, 71, 31, 52, 37, 6, 59, 157, 83, 100, 47, 94, 222, 30, 132, 214, 19, 8, 26, 250, 92, 34, 208, 81, 40, 91, 214, 59, 148, 59, 86, 93, 137, 138, 5, 104, 84, 19, 229, 60, 60, 108, 101, 37, 255, 31, 227, 78, 61, 220, 112, 240, 213, 100, 80, 253, 164, 139, 161, 46, 16, 78, 157, 235, 159, 184, 24, 129, 225, 196, 189, 242, 93, 146, 71, 244, 80, 200, 101, 146, 121, 104, 231, 115, 52, 244, 65, 79, 117, 167, 80, 225, 57, 84, 110, 58, 138, 115,

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

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

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

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

A.2.2. Decoding

TOC

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

A.2.3. Validating

TOC

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

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

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

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

TOC

A.3.1. Encoding

TOC

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

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

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

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

Base64url encoding these bytes yields this Encoded JWS Header value:

```
eyJhbGciOiJFUzI1NiJ9
```

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

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

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

```
eyJhbGciOiJFUzI1NiJ9
.
eyJpc3MiOiJqb2UiLA0KICJleHAiOjEzMDA4MTkzODAsDQogImh0dHA6Ly9leGFt
cGxlLmNvbS9pc19yb290Ijp0cnVlfQ
```

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

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

The ECDSA key consists of a public part, the EC point (x, y), and a private part d. The values of the ECDSA key used in this example, presented as the byte arrays representing three 256 bit big endian integers are:

Parameter Name	Value
x	[127, 205, 206, 39, 112, 246, 196, 93, 65, 131, 203, 238, 111, 219, 75, 123, 88, 7, 51, 53, 123, 233, 239, 19, 186, 207, 110, 60, 123, 209, 84, 69]
y	[199, 241, 68, 205, 27, 189, 155, 126, 135, 44, 223, 237, 185, 238, 185, 244, 179, 105, 93, 110, 169, 11, 36, 173, 138, 70, 35, 40, 133, 136, 229, 173]
d	[142, 155, 16, 158, 113, 144, 152, 191, 152, 4, 135, 223, 31, 93, 119, 233, 203, 41, 96, 110, 190, 210, 38, 59, 95, 87, 194, 19, 223, 132, 244, 178]

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

Result	

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

Concatenating the S array to the end of the R array and base64url encoding the result produces this value for the Encoded JWS Signature (with line breaks for display purposes only):

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

A.3.2. Decoding

TOC

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

A.3.3. Validating

TOC

Since the `alg` parameter in the header is "ES256", we validate the ECDSA P-256 SHA-256 digital signature contained in the JWS Signature. If any of the validation steps fail, the JWS MUST be rejected.

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

Validating the JWS Signature is a little different from the first example. First, we base64url decode the Encoded JWS Signature as in the previous examples but we then need to split the 64 member byte array that must result into two 32 byte arrays, the first R and the second S. We then pass (x, y), (R, S) and the bytes of the ASCII representation of the JWS Secured Input to an ECDSA signature verifier that has been configured to use the P-256 curve with the SHA-256 hash function.

As explained in Section 3.4 of the JSON Web Algorithms (JWA) **[JWA]** specification, the use of the K value in ECDSA means that we cannot validate the correctness of the digital signature in the same way we validated the correctness of the HMAC. Instead, implementations MUST use an ECDSA validator to validate the digital signature.

A.4. JWS using ECDSA P-521 SHA-512

TOC

A.4.1. Encoding

TOC

The JWS Header for this example differs from the previous example because a different ECDSA curve and hash function are used. The JWS Header used is:

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

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

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

Base64url encoding these bytes yields this Encoded JWS Header value:

```
eyJhbGciOiJFUzUxMiJ9
```

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

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

Base64url encoding these bytes yields the Encoded JWS Payload value:

```
UGF5bG9hZA
```

Concatenating the Encoded JWS Header, a period character, and the Encoded JWS Payload yields this JWS Secured Input value:

```
eyJhbGciOiJFUzUxMiJ9.UGF5bG9hZA
```

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

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

The ECDSA key consists of a public part, the EC point (x, y), and a private part d. The values of the ECDSA key used in this example, presented as the byte arrays representing three 521 bit big endian integers are:

Parameter Name	Value
x	[1, 233, 41, 5, 15, 18, 79, 198, 188, 85, 199, 213, 57, 51, 101, 223, 157, 239, 74, 176, 194, 44, 178, 87, 152, 249, 52, 235, 4, 227, 198, 186, 227, 112, 26, 87, 167, 145, 14, 157, 129, 191, 54, 49, 89, 232, 235, 203, 21, 93, 99, 73, 244, 189, 182, 204, 248, 169, 76, 92, 89, 199, 170, 193, 1, 164]
y	[0, 52, 166, 68, 14, 55, 103, 80, 210, 55, 31, 209, 189, 194, 200, 243, 183, 29, 47, 78, 229, 234, 52, 50, 200, 21, 204, 163, 21, 96, 254, 93, 147, 135, 236, 119, 75, 85, 131, 134, 48, 229, 203, 191, 90, 140, 190, 10, 145, 221, 0, 100, 198, 153, 154, 31, 110, 110, 103, 250, 221, 237, 228, 200, 200, 246]
d	[1, 142, 105, 111, 176, 52, 80, 88, 129, 221, 17, 11, 72, 62, 184, 125, 50, 206, 73, 95, 227, 107, 55, 69, 237, 242, 216, 202, 228, 240, 242, 83, 159, 70, 21, 160, 233, 142, 171, 82, 179, 192, 197, 234, 196, 206, 7, 81, 133, 168, 231, 187, 71, 222, 172, 29, 29, 231, 123, 204, 246, 97, 53, 230, 61, 130]

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

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

Concatenating the S array to the end of the R array and base64url encoding the result produces this value for the Encoded JWS Signature (with line breaks for display purposes only):

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

TOC

A.4.2. Decoding

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

TOC

A.4.3. Validating

Since the `alg` parameter in the header is "ES512", we validate the ECDSA P-521 SHA-512 digital signature contained in the JWS Signature. If any of the validation steps fail, the JWS MUST be rejected.

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

Validating the JWS Signature is similar to the previous example. First, we base64url decode the Encoded JWS Signature as in the previous examples but we then need to split the 132 member byte array that must result into two 66 byte arrays, the first R and the second S. We then pass (x, y), (R, S) and the bytes of the ASCII representation of the JWS Secured Input to an ECDSA signature verifier that has been configured to use the P-521 curve with the SHA-512 hash function.

As explained in Section 3.4 of the JSON Web Algorithms (JWA) **[JWA]** specification, the use of the K value in ECDSA means that we cannot validate the correctness of the digital signature in the same way we validated the correctness of the HMAC. Instead, implementations MUST use an ECDSA validator to validate the digital signature.

TOC

A.5. Example Plaintext JWS

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

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

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

```
eyJhbGciOiJub25lIn0
```

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

```
{"iss":"joe",
 "exp":1300819380,
```

```
 "http://example.com/is_root":true}
```

The Encoded JWS Signature is the empty string.

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

```
eyJhbGciOiJub25lIn0
.
eyJpc3MiOiJqb2UiLA0KICJleHAiOjEzMDA4MTkzODAsDQogImh0dHA6Ly9leGFt
cGxlLmNvbS9pc19yb290Ijp0cnVlfQ
.
```

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

TOC

The string 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**.

```
-----BEGIN CERTIFICATE-----
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
b20vcmVwb3NpdG9yeTAOBgNVHQ8BAf8EBAMCAQYwDQYJKoZIhvcNAQEFBQADggE
BANKGwOy9+aG2Z+5mC6IGOgRQjhVyrEp0lVPLN8tESe8HkGsz2ZbwlFalEzAFPI
UyIXvJxwqoJKSQ3kbTJSMUA2fCENZvD117esyfxVgqwcSeIaha86ykRvOe5GPLL
5CkKSkB2XIsKd83ASe8T+5o0yGPwLPk9Qnt0hCqU7S+8MxZC9Y7lhyVJEnfzuz9
p0iRFEUOOjZv2kWzRaJBydTXRE4+uXR21aITVSzGh6O1mawGhId/dQb8vxRMDsx
uxN89txJx9OjxUUAiKEngHUuHqDTMBqLdElrRhjZkAzVvb3du6/KFUJheqwNTrZ
EjYx8WnM25sgVjOuH0aBsXBTWVU+4=
-----END CERTIFICATE-----
-----BEGIN CERTIFICATE-----
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==
-----END CERTIFICATE-----
-----BEGIN CERTIFICATE-----
MIIC5zCCAlACAQEwDQYJKoZIhvcNAQEFBQAwgbsxJDAiBgNVBAcTG1ZhbGlDZXJ
0IFZhbGlkYXRpb24gTmV0d29yazEXMBUGA1UEChMOVmFsaUNlcnQsIEluYy4xNT
AzBgNVBAsTLFZhbGlDZXJ0IENsYXNzIDIgUG9saWN5IFZhbGlkYXRpb24gQXV0a
G9yaXR5MSEwHwYDVQQDExhodHRwOi8vd3d3LnZhbGljZXJ0LmNvbS8xIDAeBgkq
hkiG9w0BCQEWEWluZm9AdmFsaWNlcnQuY29tMB4XDTk5MDYyNjAwMTk1NFoXDTE
5MDYyNjAwMTk1NFowgbsxJDAiBgNVBAcTG1ZhbGlDZXJ0IFZhbGlkYXRpb24gTm
V0d29yazEXMBUGA1UEChMOVmFsaUNlcnQsIEluYy4xNTAzBgNVBAsTLFZhbGlDZ
XJ0IENsYXNzIDIgUG9saWN5IFZhbGlkYXRpb24gQXV0aG9yaXR5MSEwHwYDVQQD
ExhodHRwOi8vd3d3LnZhbGljZXJ0LmNvbS8xIDAeBgkqhkiG9w0BCQEWEWluZm9
AdmFsaWNlcnQuY29tMIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQDOOnHK5a
vIWZJV16vYdA757tn2VUdZZUcOBVXc65g2PFxTXdMwzzjsvUGJ7SVCCSRrCl6zf
N1SLUzm1NZ9WlmpZdRJEy0kTRxQb7XBhVQ7/nHk01xC+YDgkRoKWzk2Z/M/VXwb
P7RfZHM047QSv4dk+NoS/zcnwbNDu+97bi5p9wIDAQABMA0GCSqGSIb3DQEBBQU
AA4GBADt/UG9vUJSZSWI4OB9L+KXIPqeCgfYrx+jFzug6EILLGACOTb2oWH+heQ
C1u+mNr0HZDzTuIYEZoDJJKPTEjlbVUjP9UNV+mWwD5MlM/Mtsq2azSiGM5bUMM
j4QssxsodyamEwCW/POuZ6lcg5Ktz885hZo+L7tdEy8W9ViH0Pd
-----END CERTIFICATE-----
```

Appendix C. Notes on implementing base64url encoding without padding

TOC

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

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

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

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

```
  }
  return Convert.FromBase64String(s); // Standard base64 decoder
}
```

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

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

```
3 236 255 224 193
```

```
A-z_4ME
```

Appendix D. Acknowledgements

TOC

Solutions for signing JSON content were previously explored by **Magic Signatures** [MagicSignatures], **JSON Simple Sign** [JSS], and **Canvas Applications** [CanvasApp], all of which influenced this draft. Dirk Balfanz, Yaron Y. Goland, John Panzer, and Paul Tarjan all made significant contributions to the design of this specification.

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

Appendix E. Document History

TOC

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

-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. to
- 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

Nat Sakimura
Nomura Research Institute
Email: n-sakimura@nri.co.jp