JOSE Working Group	M. Jones
Internet-Draft	Microsoft
Intended status: Standards Track	J. Bradley
Expires: November 13, 2012	Ping Identity
	N. Sakimura
	NRI
	May 12, 2012

JSON Web Signature (JWS)
draft-ietf-jose-json-web-signature-02

Abstract

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

Requirements Language

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 **RFC 2119** [RFC2119].

Status of this Memo

Copyright Notice

Table of Contents

TOC

1. Introduction

JSON Web Signature (JWS) is a compact format for representing content secured with digital signatures or Message Authentication Codes (MACs) intended for space constrained environments such as HTTP Authorization headers and URI query parameters. It represents this content using JSON **[RFC4627]** data structures. The JWS digital signature and MAC mechanisms are independent of the type of content being secured, allowing arbitrary content to be secured. Cryptographic algorithms and identifiers used with this specification are enumerated in the separate JSON Web Algorithms (JWA) **[JWA]** specification. Related encryption capabilities are described in the separate JSON Web Encryption (JWE) **[JWE]** specification.

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 contents of the JWS Header and the JWS Payload.

Encoded JWS Header
: Base64url encoding of the bytes of the UTF-8 **RFC 3629** [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 Names
: The names of the members within the JSON object represented in a JWS Header.

Header Parameter Values
: The values of the members within the JSON object represented in 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.

Base64url Encoding
: For the purposes of this specification, this term always refers to 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 B** for notes on implementing base64url encoding without padding.)

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 as defined in **RFC 3986** [RFC3986].

---

3. JSON Web Signature (JWS) Overview

TOC

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.

3.1. Example JWS

TOC

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 UTF-8 representation of the JWS Secured Input (the concatenation of the Encoded JWS Header, a period ('.') character, and the Encoded JWS Payload) (which is the same as the ASCII representation) 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 **[JWA]**. 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

TOC

The `alg` (algorithm) header parameter identifies the cryptographic algorithm used to secure the JWS. A list of defined `alg` values for use with JWS is presented in Section 3.1 of the JSON Web Algorithms (JWA) **[JWA]** specification. The processing of the `alg` header parameter requires that the value MUST be one that is both supported and for which there exists a key for use with that algorithm associated with the party that digitally signed or MACed the content. The `alg` value is case sensitive. Its value MUST be a string containing a StringOrURI value. This header parameter is REQUIRED.

`alg` values SHOULD either be defined in the IANA JSON Web Signature and Encryption Algorithms registry **[JWA]** or be a URI that contains a collision resistant namespace.

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

TOC

The `jku` (JWK Set URL) header parameter is an absolute URL 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) as defined in the JSON Web Key (JWK) **[JWK]** specification. The protocol used to acquire the resource MUST provide integrity protection; an HTTP GET request to retrieve the certificate MUST use TLS **RFC 2818** [RFC2818] **RFC 5246** [RFC5246]; the identity of the server MUST be validated, as per Section 3.1 of HTTP Over TLS **[RFC2818]**. This header parameter is OPTIONAL.

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

TOC

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

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

TOC

The `x5u` (X.509 URL) header parameter is an absolute URL that refers to a resource for the X.509 public key certificate or certificate chain 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 **RFC 1421** [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 **RFC 2818** [RFC2818] **RFC 5246** [RFC5246]; the identity of the server MUST be validated, as per Section 3.1 of HTTP Over TLS **[RFC2818]**. This header parameter is OPTIONAL.

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

TOC

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 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 **[JWA]**.

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 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 (not base64url encoded) DER/BER 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.

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 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 contents of the kid parameter is unspecified. Its value MUST be a string. This header parameter is OPTIONAL.

TOC

4.1.8. "typ" (Type) Header Parameter

The typ (type) header parameter is used to declare the type of the secured content. The type value JWS MAY be used to indicate that the secured content is a JWS. The typ value is case sensitive. Its value MUST be a string. This header parameter is OPTIONAL.

MIME Media Type **RFC 2045** [RFC2045] values MAY be used as typ values.

typ values SHOULD either be defined in the IANA JSON Web Signature and Encryption "typ" Values registry **[JWA]** or be a URI that contains a collision resistant namespace.

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 defined in the IANA JSON Web Signature and Encryption Header Parameters registry **[JWA]** 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.

---

5. Rules for Creating and Validating a JWS

TOC

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 contents of the JWS Header and the JWS Payload. The JSON Web Algorithms (JWA) **[JWA]** specification enumerates a set of cryptographic algorithms and identifiers to be used with this specification. Specifically, Section 3.1 enumerates 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

7.1. Registration of application/jws MIME Media Type

TOC

This specification registers the `application/jws` MIME Media Type **RFC 2045** [RFC2045].

Type name:
  application
Subtype name:
  jws
Required parameters:
  n/a
Optional parameters:
  n/a
Encoding considerations:
  n/a
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
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

7.2. Registration of "JWS" Type Value

TOC

This specification registers the following `typ` header parameter value in the JSON Web Signature and Encryption "typ" Values registry established by the JSON Web Algorithms (JWA) **[JWA]** specification:

"typ" header parameter value:
: "JWS"

Abbreviation for MIME type:
: application/jws

Change controller:
: IETF

Description:
: [[this document]]

---

8. Security Considerations

TOC

8.1. Cryptographic Security Considerations

TOC

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.

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.

TBD: We need to also put in text about: Importance of keeping secrets secret. Rotating keys. Strengths and weaknesses of the different algorithms.

TBD: Write security considerations about the implications of using a SHA-1 hash (for compatibility reasons) for the `x5t` (x.509 certificate thumbprint).

TBD: We need a section on generating randomness in browsers; it's easy to screw up.

8.2. JSON Security Considerations

TOC

TBD: We need to look into any issues relating to security and JSON parsing. One wonders just how secure most JSON parsing libraries are. Were they ever hardened for security scenarios? If not, what kind of holes does that open up? We need to put in text about why strict JSON validation is necessary - basically, that if malformed JSON is received then the intent of the sender is impossible to reliably discern.

8.3. Unicode Comparison Security Considerations

TOC

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 and Things To Be Done (TBD)

The following items remain to be done in this draft:

- Add an example in which the payload is not a base64url encoded JSON object.
- Finish the Security Considerations section.

TOC

10. References

TOC

10.1. Normative References

[JWA] Jones, M., "JSON Web Algorithms (JWA)," May 2012.

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

[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).

[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).

[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).

[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.

TOC

10.2. 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)," May 2012.

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

[JWT] Jones, M., Balfanz, D., Bradley, J., Goland, Y., Panzer, J., Sakimura, N., and P. Tarjan, "JSON Web Token (JWT)," May 2012.

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

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

[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).

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

TOC

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 UTF-8 representation of the JWS Secured Input (which is the same as the ASCII representation) 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 UTF-8 representation of the JWS Secured Input (which is the same as the ASCII representation) 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
```

A.1.2. Decoding

TOC

Decoding the JWS first requires removing the base64url encoding from the Encoded JWS Header, the Encoded JWS Payload, and the Encoded JWS Signature. We base64url decode the inputs and turn them into the corresponding byte arrays. We decode the Encoded JWS Header byte array containing the UTF-8 representation of the JWS Header into the JWS Header string.

A.1.3. Validating

TOC

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 UTF-8 representation of the JWS Secured Input (which is the same as the ASCII representation) as input to the HMAC SHA-256 function and then taking the output and determining if it matches the JWS Signature. If it matches exactly, the HMAC has been validated.

A.2. JWS using RSA SHA-256

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 UTF-8 representation of the JWS Secured Input (which is the same as the ASCII representation) 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 (n, e), and a private exponent d. The values of the RSA key used in this example, presented as the byte arrays representing big endian integers are:

Parameter Name	Value
n	[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]
e	[1, 0, 1]
d	[18, 174, 113, 164, 105, 205, 10, 43, 195, 126, 82, 108, 69, 0, 87, 31, 29, 97, 117, 29, 100, 233, 73, 112, 123, 98, 89, 15, 157, 11, 165, 124, 150, 60, 64, 30, 63, 207, 47, 44, 211, 189, 236, 136, 229, 3, 191, 198, 67, 155, 11, 40, 200, 47, 125, 55, 151, 103, 31, 82, 19, 238, 216, 193, 90, 37, 216, 213, 206, 160, 2, 94, 227, 171, 46, 139, 127, 121, 33, 111, 198, 59, 234, 86, 39, 83, 180, 6, 68, 198, 161, 81, 39, 217, 178, 149, 69, 64, 160, 187, 225, 163, 5, 86, 152, 45, 78, 159, 222, 95, 100, 37, 241, 77, 75, 113, 52, 65, 181, 93, 199, 59, 155, 74, 237, 204, 146, 172, 227, 146, 126, 55, 245, 125, 12, 253, 94, 117, 129, 250, 81, 44, 143, 73, 97, 169, 235, 11, 128, 248, 168, 7, 70, 114, 138, 85, 255, 70, 71, 31, 52, 37, 6, 59, 157, 83, 100, 47, 94, 222, 30, 132, 214, 19, 8, 26, 250, 92, 34, 208, 81, 40, 91, 214, 59, 148, 59, 86, 93, 137, 138, 5, 104, 84, 19, 229, 60, 60, 108, 101, 37, 255, 31, 227, 78, 61, 220, 112, 240, 213, 100, 80, 253, 164, 139, 161, 46, 16, 78, 157, 235, 159, 184, 24, 129, 225, 196, 189, 242, 93, 146, 71, 244, 80, 200, 101, 146, 121, 104, 231, 115, 52, 244, 65, 79, 117, 167, 80, 225, 57, 84, 110, 58, 138, 115, 157]

The RSA private key (n, d) is then passed to the RSA signing function, which also takes the hash type, SHA-256, and the bytes of the UTF-8 representation of the JWS Secured Input (which is the same as the ASCII representation) as inputs. The result of the digital signature is a byte array S, which represents a big endian integer. In this example, S is:

Result Name	Value
S	[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 from this example requires processing the Encoded JWS Header and Encoded JWS Payload exactly as done in the first example.

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 UTF-8 representation of the JWS Secured Input (which is the same as the ASCII representation) 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 UTF-8 representation of the JWS Secured Input (which is the same as the ASCII representation) 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 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 UTF-8 representation of the JWS Secured Input (which is the same as the ASCII representation) as inputs. The result of the digital signature is the EC point (R, S), where R and S are unsigned integers. In this example, the R and S values, given as byte arrays representing big endian integers are:

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

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


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


A.3.2. Decoding

TOC

Decoding the JWS from this example requires processing the Encoded JWS Header and Encoded JWS Payload exactly as done in the first example.

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 UTF-8 representation of the JWS Secured Input (which is the same as the ASCII representation) 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. 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. Notes on implementing base64url encoding without padding

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

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

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

static byte [] base64urldecode(string arg)
{
  string s = arg;
  s = s.Replace('-', '+'); // 62nd char of encoding
  s = s.Replace('_', '/'); // 63rd char of encoding
  switch (s.Length % 4) // Pad with trailing '='s
```

```
  {
    case 0: break; // No pad chars in this case
    case 2: s += "=="; break; // Two pad chars
    case 3: s += "="; break; // One pad char
    default: throw new System.Exception(
      "Illegal base64url string!");
  }
  return Convert.FromBase64String(s); // Standard base64 decoder
}
```

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

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

```
3 236 255 224 193
```

```
A-z_4ME
```

TOC

Appendix C. Acknowledgements

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

TOC

Appendix D. Document History

-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".

TOC

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