Network Working Group	M. Jones
Internet-Draft Intended status: Standards Track Expires: May 2, 2012	Microsoft D. Balfanz Google
	J. Bradley independent Y. Goland
	Microsoft J. Panzer Google
	N. Sakimura Nomura Research Institute
	P. Tarjan Facebook October 30, 2011

TOC

JSON Web Signature (JWS)
draft-jones-json-web-signature-03

Abstract

JSON Web Signature (JWS) is a means of representing signed content using JSON data structures. 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].

Table of Contents

TOC

1. Introduction

JSON Web Signature (JWS) is a compact signature format intended for space constrained environments such as HTTP Authorization headers and URI query parameters. It represents signed content using JSON **[RFC4627]** data structures. The JWS signature mechanisms are independent of the type of content being signed, allowing arbitrary content to be signed. A related encryption capability is described in a 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.

JWS Header
: A string containing a JSON object that describes the signature applied to the JWS Header and the JWS Payload to create the JWS Signature.

JWS Payload
: The bytes to be signed - a.k.a., the message.

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 Signing 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.

Digital Signature
: For the purposes of this specification, we use this term to encompass both Hash-based Message Authentication Codes (HMACs), which can provide authenticity but not non-repudiation, and digital signatures using public key algorithms, which can provide both. Readers should be aware of this distinction, despite the decision to use a single term for both concepts to improve readability of the specification.

Base64url Encoding
: For the purposes of this specification, this term always refers to the he 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.)

TOC

3. JSON Web Signature (JWS) Overview

JWS represents signed content using JSON data structures and base64url encoding. The representation consists of three parts: the JWS Header, the JWS Payload, and the JWS Signature. The three parts are base64url-encoded for transmission, and typically represented as the concatenation of the encoded strings in that order, with the three strings being separated by period ('.') characters, as is done when used in JSON Web Tokens (JWTs) **[JWT]**.

A base64url encoded JSON object - the JWS Header - describes the signature method used. A portion of the base64url encoded content that is signed is the Encoded JWS Payload. Finally, JWSs contain a signature that ensures the integrity of the contents of the JWS Header and the JWS Payload. This signature value is base64url encoded to produce the Encoded JWS Signature.

The member names within the JWS Header are referred to as Header Parameter Names. These names MUST be unique. The corresponding values are referred to as Header Parameter Values. The JWS Header MUST contain an `alg` parameter, the value of which is a string that unambiguously identifies the algorithm used to sign the JWS Header and the JWS Payload to produce the JWS Signature.

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 signed 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.


```
eyJpc3MiOiJqb2UiLA0KICJleHAiOjEzMDA4MTkzODAsDQogImh0dHA6Ly9leGFtcGxlLmNvbS9pc19yb290IjpOcnVlfQ
```


Signing the UTF-8 representation of the JWS Signing Input (the concatenation of the Encoded JWS Header, a period ('.') character, and the Encoded JWS Payload) with the HMAC SHA-256 algorithm and base64url encoding the result, as per **Section 6.1**, yields this Encoded JWS Signature value:


```
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 signature applied to the Encoded JWS Header and the Encoded JWS Payload and optionally additional properties of the JWS. Implementations MUST understand the entire contents of the header; otherwise, the JWS MUST be rejected for processing.

TOC

4.1. Reserved Header Parameter Names

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

Header Parameter Name	JSON Value Type	Header Parameter Syntax	Header Parameter Semantics
alg	string	StringOrURI	The `alg` (algorithm) header parameter identifies the cryptographic algorithm used to secure the JWS. A list of reserved alg values is in **Table 3**. The processing of the `alg` (algorithm) header parameter, if present, requires that the value of the `alg` header parameter MUST be one that is both supported and for which there exists a key for use with that algorithm associated with the signer of the content. The `alg` parameter value is case sensitive. This header parameter is REQUIRED.
typ	string	String	The `typ` (type) header parameter is used to declare the type of the signed content. The `typ` value is case sensitive. This header parameter is OPTIONAL.
			The `jku` (JSON Web Key URL) header parameter is an absolute

jku	string	URL	URL that refers to a resource for a set of JSON-encoded public keys, one of which corresponds to the key that was used to sign the JWS. The keys MUST be encoded as described 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] with server authentication **RFC 6125** [RFC6125]. This header parameter is OPTIONAL.
kid	string	String	The `kid` (key ID) header parameter is a hint indicating which specific key owned by the signer should be used to validate the signature. This allows signers to explicitly signal a change of key to recipients. The interpretation of the contents of the `kid` parameter is unspecified. This header parameter is OPTIONAL.
x5u	string	URL	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 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 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] with server authentication **RFC 6125** [RFC6125]. This header parameter is OPTIONAL.
x5t	string	String	The `x5t` (x.509 certificate thumbprint) header parameter provides a base64url encoded SHA-1 thumbprint (a.k.a. digest) of the DER encoding of an X.509 certificate that can be used to match the certificate. This header parameter is OPTIONAL.

Table 1: Reserved Header Parameter Definitions

Additional reserved header parameter names MAY be defined via the IANA JSON Web Signature Header Parameters registry, as per **Section 7**. The syntax values used above are defined as follows:

Syntax Name	Syntax Definition
IntDate	The number of seconds from 1970-01-01T0:0:0Z as measured in UTC until the desired date/time. See **RFC 3339** [RFC3339] for details regarding date/times in general and UTC in particular.
String	Any string value MAY be used.
StringOrURI	Any string value MAY be used but a value containing a ":" character MUST be a URI as defined in **RFC 3986** [RFC3986].
URL	A URL as defined in **RFC 1738** [RFC1738].

Table 2: Header Parameter Syntax Definitions

4.2. Public Header Parameter Names

TOC

Additional header parameter names can be defined by those using JWSs. However, in order to prevent collisions, any new header parameter name or algorithm value SHOULD either be defined in the IANA JSON Web Signature Header Parameters registry or be defined as a URI that contains a collision resistant namespace. In each case, the definer of the name or value MUST take reasonable precautions to make sure they are in control of the part of the namespace they use to define the header parameter name.

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

4.3. Private Header Parameter Names

TOC

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.

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

5. Rules for Creating and Validating a JWS

TOC

To create a JWS, one MUST follow these steps:

1. Create the content to be used as the JWS Payload.
2. Base64url encode the JWS Payload. This encoding becomes the Encoded JWS Payload.
3. Create a JSON object containing a set of desired header parameters. Note that white space is explicitly allowed in the representation and no canonicalization is performed before encoding.
4. Translate this JSON object's Unicode code points into UTF-8, as defined in **RFC 3629** [RFC3629].
5. Base64url encode the UTF-8 representation of this JSON object as defined in this specification (without padding). This encoding becomes the Encoded JWS Header.
6. Compute the JWS Signature in the manner defined for the particular algorithm being used. The JWS Signing Input is always the concatenation of the Encoded JWS Header, a period ('.') character, and the Encoded JWS Payload. The `alg` header parameter MUST be present in the JSON Header, with the algorithm value accurately representing the algorithm used to construct the JWS Signature.
7. Base64url encode the representation of the JWS Signature to create the Encoded JWS Signature.

When validating a JWS, the following steps MUST be taken. If any of the listed steps fails, then the signed content MUST be rejected.

1. The Encoded JWS Payload MUST be successfully base64url decoded following the restriction given in this specification that no padding characters have been used.
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 JWS Header MUST be completely valid JSON syntax conforming to **RFC 4627** [RFC4627].
4. The Encoded JWS Signature MUST be successfully base64url decoded following the restriction given in this specification that no padding characters have been used.
5. The JWS Header MUST be validated to only include parameters and values whose syntax and semantics are both understood and supported.
6. The JWS Signature MUST be successfully validated against the JWS Header and JWS Payload in the manner defined for the algorithm being used, which MUST be accurately represented by the value of the `alg` 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. Comparing Unicode strings, however, has significant security implications, as per **Section 8**.

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. Signing JWSs with Cryptographic Algorithms

TOC

JWSs use specific cryptographic algorithms to sign the contents of the JWS Header and the JWS Payload. The use of the following algorithms for producing JWSs is defined in this section. The table below is the list of `alg` header parameter values reserved by this specification, each of which is explained in more detail in the following sections:

Alg Parameter Value	Algorithm
HS256	HMAC using SHA-256 hash algorithm
HS384	HMAC using SHA-384 hash algorithm
HS512	HMAC using SHA-512 hash algorithm
RS256	RSA using SHA-256 hash algorithm
RS384	RSA using SHA-384 hash algorithm
RS512	RSA using SHA-512 hash algorithm
ES256	ECDSA using P-256 curve and SHA-256 hash algorithm
ES384	ECDSA using P-384 curve and SHA-384 hash algorithm
ES512	ECDSA using P-521 curve and SHA-512 hash algorithm

Table 3: JSON Web Signature Reserved Algorithm Values

See **Appendix B** for a table cross-referencing the `alg` values used in this specification with the equivalent identifiers used by other standards and software packages.

Of these algorithms, only HMAC SHA-256 MUST be implemented by conforming implementations. It is RECOMMENDED that implementations also support the RSA SHA-256 and ECDSA P-256 SHA-256 algorithms. Support for other algorithms and key sizes is OPTIONAL.

The signed content for a JWS is the same for all algorithms: the concatenation of the Encoded JWS Header, a period ('.') character, and the Encoded JWS Payload. This character sequence is referred to as the JWS Signing Input. Note that if the JWS represents a JWT, this corresponds to the portion of the JWT representation preceding the second period character. The UTF-8 representation of the JWS Signing Input is passed to the respective signing algorithms.

6.1. Creating a JWS with HMAC SHA-256, HMAC SHA-384, or HMAC SHA-512

TOC

Hash based Message Authentication Codes (HMACs) enable one to use a secret plus a cryptographic hash function to generate a Message Authentication Code (MAC). This can be used to demonstrate that the MAC matches the hashed content, in this case the JWS Signing Input, which therefore demonstrates that whoever generated the MAC was in possession of the secret. The means of exchanging the shared key is outside the scope of this specification.

The algorithm for implementing and validating HMACs is provided in **RFC 2104** [RFC2104]. This section defines the use of the HMAC SHA-256, HMAC SHA-384, and HMAC SHA-512 cryptographic hash functions as defined in **FIPS 180-3** [FIPS.180-3]. The reserved `alg` header parameter values `HS256`, `HS384`, and `HS512` are used in the JWS Header to indicate that the Encoded JWS Signature contains a base64url encoded HMAC value using the respective hash function.

The HMAC SHA-256 MAC is generated as follows:

1. Apply the HMAC SHA-256 algorithm to the UTF-8 representation of the JWS Signing Input using the shared key to produce an HMAC.
2. Base64url encode the HMAC, as defined in this specification.

The output is the Encoded JWS Signature for that JWS.

The HMAC SHA-256 MAC for a JWS is validated as follows:

1. Apply the HMAC SHA-256 algorithm to the UTF-8 representation of the JWS Signing Input of the JWS using the shared key.
2. Base64url encode the previously generated HMAC, as defined in this specification.
3. If the JWS Signature and the previously calculated value exactly match, then one has confirmation that the key was used to generate the HMAC on the JWS and that the contents of the JWS have not be tampered with.
4. If the validation fails, the signed content MUST be rejected.

Signing with the HMAC SHA-384 and HMAC SHA-512 algorithms is performed identically to the procedure for HMAC SHA-256 - just with correspondingly longer key and result values.

6.2. Creating a JWS with RSA SHA-256, RSA SHA-384, or RSA SHA-512

TOC

This section defines the use of the RSASSA-PKCS1-v1_5 signature algorithm as defined in **RFC 3447** [RFC3447], Section 8.2 (commonly known as PKCS#1), using SHA-256, SHA-384, or SHA-512 as the hash function. The RSASSA-PKCS1-v1_5 algorithm is described in **FIPS 186-3** [FIPS.186-3], Section 5.5, and the SHA-256, SHA-384, and SHA-512 cryptographic hash functions are defined in **FIPS 180-3** [FIPS.180-3]. The reserved `alg` header parameter values `RS256`, `RS384`, and `RS512` are used in the JWS Header to indicate that the Encoded JWS Signature contains a base64url encoded RSA signature using the respective hash function.

The public keys employed may be retrieved using Header Parameter methods described in **Section 4.1** or may be distributed using methods that are outside the scope of this specification.

A 2048-bit or longer key length MUST be used with this algorithm.

The RSA SHA-256 signature is generated as follows:

1. Generate a digital signature of the UTF-8 representation of the JWS Signing Input using RSASSA-PKCS1-V1_5-SIGN and the SHA-256 hash function with the desired private key. The output will be a byte array.
2. Base64url encode the byte array, as defined in this specification.

The output is the Encoded JWS Signature for that JWS.

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

1. Take the Encoded JWS Signature and base64url decode it into a byte array. If decoding fails, the signed content MUST be rejected.
2. Submit the UTF-8 representation of the JWS Signing Input and the public key corresponding to the private key used by the signer to the RSASSA-PKCS1-V1_5-VERIFY algorithm using SHA-256 as the hash function.
3. If the validation fails, the signed content MUST be rejected.

Signing with the RSA SHA-384 and RSA SHA-512 algorithms is performed identically to the procedure for RSA SHA-256 - just with correspondingly longer key and result values.

6.3. Creating a JWS with ECDSA P-256 SHA-256, ECDSA P-384 SHA-384, or ECDSA P-521 SHA-512

TOC

The Elliptic Curve Digital Signature Algorithm (ECDSA) is defined by **FIPS 186-3** [FIPS.186-3]. ECDSA provides for the use of Elliptic Curve cryptography, which is able to provide equivalent security to RSA cryptography but using shorter key lengths and with greater processing speed. This means that ECDSA signatures will be substantially smaller in terms of length than equivalently strong RSA Digital Signatures.

This specification defines the use of ECDSA with the P-256 curve and the SHA-256 cryptographic hash function, ECDSA with the P-384 curve and the SHA-384 hash function, and ECDSA with the P-521 curve and the SHA-512 hash function. The P-256, P-384, and P-521 curves are also defined in FIPS 186-3. The reserved `alg` header parameter values `ES256`, `ES384`, and `ES512` are used in the JWS Header to indicate that the Encoded JWS Signature contains a base64url encoded ECDSA P-256 SHA-256, ECDSA P-384 SHA-384, or ECDSA P-521 SHA-512 signature, respectively.

The public keys employed may be retrieved using Header Parameter methods described in **Section 4.1** or may be distributed using methods that are outside the scope of this specification.

A JWS is signed with an ECDSA P-256 SHA-256 signature as follows:

1. Generate a digital signature of the UTF-8 representation of the JWS Signing Input using ECDSA P-256 SHA-256 with the desired private key. The output will be the EC point (R, S), where R and S are unsigned integers.
2. Turn R and S into byte arrays in big endian order. Each array will be 32 bytes long.
3. Concatenate the two byte arrays in the order R and then S.
4. Base64url encode the 64 byte array, as defined in this specification.

The output is the Encoded JWS Signature for the JWS.

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

1. Take the Encoded JWS Signature and base64url decode it into a byte array. If decoding fails, the signed content MUST be rejected.
2. The output of the base64url decoding MUST be a 64 byte array.
3. Split the 64 byte array into two 32 byte arrays. The first array will be R and the second S. Remember that the byte arrays are in big endian byte order; please check the ECDSA validator in use to see what byte order it requires.
4. Submit the UTF-8 representation of the JWS Signing Input, R, S and the public key (x, y) to the ECDSA P-256 SHA-256 validator.
5. If the validation fails, the signed content MUST be rejected.

The ECDSA validator will then determine if the digital signature is valid, given the inputs. Note that ECDSA digital signature contains a value referred to as K, which is a random number generated for each digital signature instance. This means that two ECDSA digital signatures using exactly the same input parameters will output different signatures because their K values will be different. The consequence of this is that one must validate an ECDSA signature by submitting the previously specified inputs to an ECDSA validator.

Signing with the ECDSA P-384 SHA-384 and ECDSA P-521 SHA-512 algorithms is performed identically to the procedure for ECDSA P-256 SHA-256 - just with correspondingly longer key and result values.

6.4. Additional Algorithms

Additional algorithms MAY be used to protect JWSs with corresponding `alg` header parameter values being defined to refer to them. New `alg` header parameter values SHOULD either be defined in the IANA JSON Web Signature Algorithms registry or be a URI that contains a collision resistant namespace. In particular, the use of algorithm identifiers defined in **XML DSIG** [RFC3275] and related specifications is permitted.

7. IANA Considerations

This specification calls for:

- A new IANA registry entitled "JSON Web Signature Header Parameters" for reserved header parameter names is defined in **Section 4.1**. Inclusion in the registry is RFC Required in the **RFC 5226** [RFC5226] sense for reserved JWS header parameter names that are intended to be interoperable between implementations. The registry will just record the reserved header parameter name and a pointer to the RFC that defines it. This specification defines inclusion of the header parameter names defined in **Table 1**.
- A new IANA registry entitled "JSON Web Signature Algorithms" for reserved values used with the `alg` header parameter values is defined in **Section 6.4**. Inclusion in the registry is RFC Required in the **RFC 5226** [RFC5226] sense. The registry will just record the `alg` value and a pointer to the RFC that defines it. This specification defines inclusion of the algorithm values defined in **Table 3**.

8. Security Considerations

TBD: Lots of work to do here. We need to remember 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? Also, we need to walk through the JSON standard and see what kind of issues we have especially around comparison of names. For instance, comparisons of header parameter names and other parameters must occur after they are unescaped. Need to also put in text about: Importance of keeping secrets secret. Rotating keys. Strengths and weaknesses of the different algorithms.

TBD: 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. One example of malformed JSON that MUST be rejected is an object in which the same member name occurs multiple times.

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

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

8.1. Unicode Comparison Security Issues

Header parameter names in JWSs are Unicode strings. For security reasons, the representations of these names must be compared verbatim after performing any escape processing (as per **RFC 4627** [RFC4627], Section 2.5).

This means, for instance, that these JSON strings must compare as being equal ("sig", "\u0073ig"), whereas these must all compare as being not equal to the first set or to each other ("SIG", "Sig", "si\u0047").

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

9. Open Issues and Things To Be Done (TBD)

The following items remain to be done in this draft:

- Consider whether there is a better term than "Digital Signature" for the concept that includes both HMACs and digital signatures using public keys.
- Consider whether we really want to allow private header parameter names that are not registered with IANA and are not in collision-resistant namespaces. Eventually this could result in interop nightmares where you need to have different code to talk to different endpoints that "knows" about each endpoint's private parameters.
- Clarify the optional ability to provide type information in the JWS header. Specifically, clarify the intended use of the `typ` Header Parameter, whether it conveys syntax or semantics, and indeed, whether this is the right approach. Also clarify the relationship between these type values and **MIME** [RFC2045] types.
- Clarify the semantics of the `kid` (key ID) header parameter. Open issues include: What happens if a `kid` header is received with an unrecognized value? Is that an error? Should it be treated as if it's empty? What happens if the header has a recognized value but the value doesn't match the key associated with that value, but it does match another key that is associated with the issuer? Is that an error?
- Consider whether a key type parameter should also be introduced.
- Since RFC 3447 Section 8 explicitly calls for people NOT to adopt RSASSA-PKCS1 for new applications and instead requests that people transition to RSASSA-PSS, we probably need some Security Considerations text explaining why RSASSA-PKCS1 is being used (it's what's commonly implemented) and what the potential consequences are.
- Add Security Considerations text on timing attacks.
- It would be good to have a confirmation method element so it could be used with holder-of-key.
- Think about how to best describe the concept currently described as "the bytes of the UTF-8 representation of". Possible terms to use instead of "bytes of" include "byte sequence", "octet series", and "octet sequence". Also consider whether we want to add an overall clarifying statement somewhere in each spec something like "every place we say 'the UTF-8 representation of X', we mean 'the bytes of the UTF-8 representation of X'". That would potentially allow us to omit the "the bytes of" part everywhere else.
- Consider whether a media type should be proposed, such as "application/jws".
- Finish the Security Considerations section.
- Add an example in which the payload is not a base64url encoded JSON object.
- Consider having an algorithm that is a MAC using SHA-256 that provides content integrity but for which there is no associated secret. This would be like the JWT "alg":"none", in that no validation of the authenticity content is provided, but with a checksum provided.
- Consider whether to define the JWT "alg":"none" here, rather than in the JWT spec.

10. References

10.1. Normative References

[FIPS.180-3] National Institute of Standards and Technology, "Secure Hash Standard (SHS)," FIPS PUB 180-3, October 2008.

[FIPS.186-3] National Institute of Standards and Technology, "Digital Signature Standard (DSS)," FIPS PUB 186-3, June 2009.

[JWK] Jones, M., "JSON Web Key (JWK)," October 2011.

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

[RFC1738] Berners-Lee, T., Masinter, L., and M. McCahill, "Uniform Resource Locators (URL)," RFC 1738, December 1994 (TXT).

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

[RFC2104] Krawczyk, H., Bellare, M., and R. Canetti, "HMAC: Keyed-Hashing for Message Authentication," RFC 2104, February 1997 (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).

[RFC3339] Klyne, G., Ed. and C. Newman, "Date and Time on the Internet: Timestamps," RFC 3339, July 2002 (TXT, HTML, XML).

[RFC3447] Jonsson, J. and B. Kaliski, "Public-Key Cryptography Standards (PKCS) #1: RSA Cryptography Specifications Version 2.1," RFC 3447, February 2003 (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).

[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).

[RFC5785] Nottingham, M. and E. Hammer-Lahav, "Defining Well-Known Uniform Resource Identifiers (URIs)," RFC 5785, April 2010 (TXT).

[RFC6125] Saint-Andre, P. and J. Hodges, "Representation and Verification of Domain-Based Application Service Identity within Internet Public Key Infrastructure Using X.509 (PKIX) Certificates in the Context of Transport Layer Security (TLS)," RFC 6125, March 2011 (TXT).

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

10.2. Informative References

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

[JCA] Oracle, "Java Cryptography Architecture," 2011.

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

[JWE] Jones, M., Bradley, J., and N. Sakimura, "JSON Web Encryption (JWE)," October 2011.

[JWT] Jones, M., Balfanz, D., Bradley, J., Goland, Y., Panzer, J., Sakimura, N., and P. Tarjan, "JSON Web Token (JWT)," October 2011.

[MagicSignatures] Panzer (editor), J., Laurie, B., and D. Balfanz, "Magic Signatures," August 2010.

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

Appendix A. JWS Examples

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

A.1. JWS using HMAC SHA-256

A.1.1. Encoding

The following example JWS Header declares that the data structure is a JSON Web Token (JWT) **[JWT]** and the JWS Signing Input is signed using the HMAC SHA-256 algorithm. Note that white space is explicitly allowed in JWS Header strings and no canonicalization is performed before encoding.

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

The following byte array contains the UTF-8 characters for 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 this UTF-8 representation yields this Encoded JWS Header value:


```
eyJ0eXAiOiJKV1QiLA0KICJhbGciOiJIUzI1NiJ9
```


The JWS Payload used in this example follows. (Note that the payload can be any base64url encoded content, and need not be a base64url encoded JSON object.)

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

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

The following byte array contains the UTF-8 characters for 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:


```
eyJpc3MiOiJqb2UiLA0KICJleHAiOjEzMDA4MTkzODAsDQogImh0dHA6Ly9leGFtcGxlLmNvbS9pc19yb290Ijp0cnVlfQ
```


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


```
eyJ0eXAiOiJKV1QiLA0KICJhbGciOiJIUzI1NiJ9
.
eyJpc3MiOiJqb2UiLA0KICJleHAiOjEzMDA4MTkzODAsDQogImh0dHA6Ly9leGFtcGxlLmNvbS9pc19yb290Ijp0cnVlfQ
```


The UTF-8 representation of the JWS Signing 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 UTF-8 representation of the JWS Signing 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
```


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 translate the header input byte array containing UTF-8 encoded characters 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 signature contained in the JWS Signature. If any of the validation steps fail, the signed content MUST be rejected.

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

To validate the signature, we repeat the previous process of using the correct key and the UTF-8 representation of the JWS Signing Input as input to a SHA-256 HMAC function and then taking the output and determining if it matches the JWS Signature. If it matches exactly, the signature has been validated.

A.2. JWS using RSA SHA-256

TOC

A.2.1. Encoding

TOC

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 signature algorithm employed.) The JWS Header used is:

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

The following byte array contains the UTF-8 characters for the JWS Header:

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

Base64url encoding this UTF-8 representation 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 Signing Input value (with line breaks for display purposes only):


```
eyJhbGciOiJSUzI1NiJ9
.
eyJpc3MiOiJqb2UiLA0KICJleHAiOjEzMDA4MTkzODAsDQogImh0dHA6Ly9leGFtcGxlLmNvbS9pc19yb290Ijp0cnVlfQ
```


The UTF-8 representation of the JWS Signing 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 (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 UTF-8 representation of the JWS Signing Input as inputs. The result of the 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 signature produces this value for the Encoded JWS Signature:


```
cC4hiUPoj9Eetdgtv3hF80EGrhuB__dzERat0XF9g2VtQgr9PJbu3XOiZj5RZmh7AAuHIm4Bh-0Qc_lF5YKt_O8W2Fp5jujGbds9uJdbF9CUAr7t1dnZcAcQjbKBYNX4BAynRFdiuB--f_nZLgrnbyTyWzO75vRK5h6xBArLIARNPvkSjtQBMHlb1L07Qe7K0GarZRmB_eSN9383LcOLn6_dO--xi12jzDwusC-eOkHWEsqtFZESc6BfI7noOPqvhJ1phCnvWh6IeYI2w9QOYEUipUTI8np6LbgGY9Fs98rqVt5AXLIhWkWywlVmtVrBp0igcN_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 signature contained in the JWS Signature. If any of the validation steps fail, the signed content 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 signature S to check. We then pass (n, e), S and the UTF-8 representation of the JWS Signing 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 characters for the JWS Header:

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

Base64url encoding this UTF-8 representation 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 Signing Input value (with line breaks for display purposes only):


```
eyJhbGciOiJFUzI1NiJ9
.
eyJpc3MiOiJqb2UiLA0KICJleHAiOjEzMDA4MTkzODAsDQogImh0dHA6Ly9leGFtcGxlLmNvbS9pc19yb290Ijp0cnVlfQ
```


The UTF-8 representation of the JWS Signing 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 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 UTF-8 representation of the JWS Signing Input as inputs. The result of the 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:


```
DtEhU3ljbEg8L38VWAfUAqOyKAM6-Xx-F4GawxaepmXFCgfTjDxw5djxLa8ISlSApmWQxfKTUJqPP3-Kg6NU1Q
```


A.3.2. Decoding

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

Since the `alg` parameter in the header is "ES256", we validate the ECDSA P-256 SHA-256 signature contained in the JWS Signature. If any of the validation steps fail, the signed content 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 UTF-8 representation of the JWS Signing Input to an ECDSA signature verifier that has been configured to use the P-256 curve with the SHA-256 hash function.

As explained in **Section 6.3**, the use of the k value in ECDSA means that we cannot validate the correctness of the signature in the same way we validated the correctness of the HMAC. Instead, implementations MUST use an ECDSA validator to validate the signature.

Appendix B. Algorithm Identifier Cross-Reference

This appendix contains a table cross-referencing the `alg` values used in this specification with the equivalent identifiers used by other standards and software packages. See **XML DSIG** [RFC3275] and **Java Cryptography Architecture** [JCA] for more information about the names defined by those documents.

Algorithm	JWS	XML DSIG	JCA	OID
HMAC using SHA-256 hash algorithm	HS256	http://www.w3.org/2001/04/xmldsig-more#hmac-sha256	HmacSHA256	1.2.840.113549.2.9
HMAC using SHA-384 hash algorithm	HS384	http://www.w3.org/2001/04/xmldsig-more#hmac-sha384	HmacSHA384	1.2.840.113549.2.10
HMAC using SHA-512 hash algorithm	HS512	http://www.w3.org/2001/04/xmldsig-more#hmac-sha512	HmacSHA512	1.2.840.113549.2.11
RSA using SHA-256 hash algorithm	RS256	http://www.w3.org/2001/04/xmldsig-more#rsa-sha256	SHA256withRSA	1.2.840.113549.1.1.11
RSA using SHA-384 hash algorithm	RS384	http://www.w3.org/2001/04/xmldsig-more#rsa-sha384	SHA384withRSA	1.2.840.113549.1.1.12
RSA using SHA-512 hash algorithm	RS512	http://www.w3.org/2001/04/xmldsig-more#rsa-sha512	SHA512withRSA	1.2.840.113549.1.1.13
ECDSA using P-256 curve and SHA-256 hash algorithm	ES256	http://www.w3.org/2001/04/xmldsig-more#ecdsa-sha256	SHA256withECDSA	1.2.840.10045.3.1.7
ECDSA using P-384 curve and SHA-384 hash algorithm	ES384	http://www.w3.org/2001/04/xmldsig-more#ecdsa-sha384	SHA384withECDSA	1.3.132.0.34
ECDSA using P-521 curve and SHA-512 hash algorithm	ES512	http://www.w3.org/2001/04/xmldsig-more#ecdsa-sha512	SHA512withECDSA	1.3.132.0.35

Table 4: Algorithm Identifier Cross-Reference

Appendix C. Notes on implementing base64url encoding without padding

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

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

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

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

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

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

```
3 236 255 224 193
```

```
A-z_4ME
```

Appendix D. Acknowledgements

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

Appendix E. Document History

-03

- Simplified terminology to better match JWE, where the terms "JWS Header" and "Encoded JWS Header", are now used, for instance, rather than the previous terms "Decoded JWS Header Input" and "JWS Header Input". Likewise the terms "JWS Payload" and "JWS Signature" are now used, rather than "JWS Payload Input" and "JWS Crypto Output".
- The `jku` and `x5u` URLs are now required to be absolute URLs.
- Removed this unnecessary language from the `kid` description: "Omitting this parameter is equivalent to setting it to an empty string".
- Changed StringAndURI to StringOrURI.

-02

- Reference the JSON Web Key (JWK) specification from the `jku` header parameter.

-01

- Changed RSA SHA-256 from MUST be supported to RECOMMENDED that it be supported. Rationale: Several people have objected to the requirement for implementing RSA SHA-256, some because they will only be using HMACs and symmetric keys, and others because they only want to use ECDSA when using asymmetric keys, either for security or key length reasons, or both.
- Clarified that `x5u` is an HTTPS URL referencing a PEM-encoded certificate or certificate chain.
- Clarified that the `alg` parameter value is case sensitive.
- Changed `x5t` (x.509 certificate thumbprint) to use a SHA-1 hash, rather than a SHA-256 hash, for compatibility reasons.

-00

- Created first signature draft using content split from draft-jones-json-web-token-01. This split introduced no semantic changes.

Authors' Addresses

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

Dirk Balfanz
Google
Email: balfanz@google.com

John Bradley
independent
Email: ve7jtb@ve7jtb.com

Yaron Y. Goland
Microsoft
Email: yarong@microsoft.com

John Panzer
Google
Email: jpanzer@google.com

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

Paul Tarjan
Facebook
Email: pt@fb.com