Network Working Group	M. Jones	
Internet-Draft	Microsoft	
Intended status: Standards Track	D. Balfanz	
Expires: July 1, 2011	Google	
	J. Bradley	
	independent	
	Y. Goland	
	Microsoft	
	J. Panzer	
	Google	
	N. Sakimura	
	Nomura Research Institute	
	December 28, 2010	

TOC

JSON Web Token (JWT)
draft-jones-json-web-token-00

Abstract

JSON Web Token (JWT) defines a token format that can encode claims transferred between two parties. The claims in a JWT are encoded as a JSON object that is digitally signed.

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 (Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels," March 1997.) [RFC2119].

Status of this Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.
Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at http://datatracker.ietf.org/drafts/current/.
Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any

time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as “work in progress.”
This Internet-Draft will expire on July 1, 2011.

Table of Contents

---

1. Introduction

TOC

JSON Web Token (JWT) is a simple token format intended for space constrained environments such as HTTP Authorization headers and URI query parameters. JWTs encode the claims to be transmitted as a JSON object (as defined in RFC 4627 (Crockford, D., "The application/json Media Type for JavaScript Object Notation (JSON)," July 2006.) [RFC4627]) that is base64url encoded and digitally signed.
The suggested pronunciation of JWT is the same as the English word "jot".

---

2. Terminology

TOC

JSON Web Token (JWT) A string consisting of three JWT Token Segments: the JWT Envelope Segment, the JWT Claim Segment, and

the JWT Crypto Segment, in that order, with the segments being separated by period ('.') characters.

JWT Token Segment One of the three parts that make up a JSON Web Token (JWT). JWT Token Segments are always base64url encoded values.

JWT Envelope Segment A JWT Token Segment containing a base64url encoded JSON object that describes the signature applied to the JWT Claim Segment.

JWT Claim Segment A JWT Token Segment containing a base64url encoded JSON object that encodes the claims represented by the JWT.

JWT Crypto Segment A JWT Token Segment containing base64url encoded cryptographic signature material that secures the JWT Crypto Segment's contents.

Decoded JWT Envelope Segment A JWT Envelope Segment that has been base64url decoded back into a JSON object.

Decoded JWT Claim Segment A JWT Claim Segment that has been base64url decoded back into a JSON object.

Decoded JWT Crypto Segment A JWT Crypto Segment that has been base64url decoded back into cryptographic material.

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 (Josefsson, S., "The Base16, Base32, and Base64 Data Encodings," October 2006.) [RFC4648], Section 5, with the '=' padding characters omitted, as permitted by Section 3.2; see Section 7 (Base64url encoding as used by JWTs) for more details.

---

3. JSON Web Token (JWT) Overview

TOC

JWTs represent a set of claims as a JSON object that is base64url encoded and digitally signed. As per RFC 4627 (Crockford, D., "The application/json Media Type for JavaScript Object Notation (JSON)," July 2006.) [RFC4627] Section 2.2, the JSON object consists of zero or more name/value pairs (or members), where the names are strings and the values are arbitrary JSON values. These members are the claims represented by the JWT. The JSON object is base64url encoded to produce

the JWT Claim Segment. An accompanying base64url encoded JSON envelope object describes the signature method used.
The names within the object MUST be unique. The names within the JSON object are referred to as Claim Names. The corresponding values are referred to as Claim Values.
JWTs contain a signature that ensures the integrity of the content of the JSON Claim Segment. This signature value is carried in the JWT Crypto Segment. The JSON Envelope object MUST contain an "alg" parameter, the value of which is a string that unambiguously identifies the algorithm used to sign the JWT Claim Segment to produce the JWT Crypto Segment.

3.1. Example JWT

TOC

The following is an example of a JSON object that can be encoded to produce a JWT:

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

Base64url encoding the UTF-8 representation of the JSON object yields this JWT Claim Segment value:

```
eyJpc3MiOiJqb2UiLA0KICJleHAiOjEzMDA4MTkzODAsDQogImh0dHA6Ly9leGFtcGxlLmNvbS9pc19yb290IjpO
```

The following example JSON envelope object declares that the encoded object is a JSON Web Token (JWT) and the JWT Claim Segment is signed using the HMAC SHA-256 algorithm:

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

Base64url encoding the UTF-8 representation of the JSON envelope object yields this JWT Envelope Segment value:

```
eyJ0eXAiOiJKV1QiLA0KICJhbGciOiJIUzI1NiJ9
```

Signing the JWT Claim Segment with the HMAC SHA-256 algorithm and base64url encoding the result, as per Section 8.1 (Signing a JWT with HMAC SHA-256), yields this JWT Crypto Segment value:

```
35usWj9X8HwGS-CDcx1JP2NmqcrLwZ4EKp8sNThf3cY
```

Combining these segments in the order Envelope.Claims.Signature with period characters between the segments yields this complete JWT (with line breaks for display purposes only):

```
eyJ0eXAiOiJKV1QiLA0KICJhbGciOiJIUzI1NiJ9
.
eyJpc3MiOiJqb2UiLA0KICJleHAiOjEzMDA4MTkzODAsDQogImh0dHA6Ly9leGFtcGxlLmNvbS9pc19yb290IjpU
.
35usWj9X8HwGS-CDcx1JP2NmqcrLwZ4EKp8sNThf3cY
```

This computation is illustrated in more detail in Section 13.1 (JWT using HMAC SHA-256).

---

4. JWT Claims

TOC

The members of the JSON object represented by the Decoded JWT Claim Segment contain the claims. Note however, that the set of claims a JWT must contain to be considered valid is context-dependent and is outside the scope of this specification.
There are three classes of JWT Claim Names: Reserved Claim Names, Public Claim Names, and Private Claim Names.

4.1. Reserved Claim Names

TOC

The following claim names are reserved. None of the claims defined in the table below are intended to be mandatory, but rather, provide a starting point for a set of useful, interoperable claims. All the names are short because a core goal of JWTs is for the tokens themselves to be short.

Claim Name	JSON Value Type	Claim Syntax	Claim Semantics
exp	integer	IntDate	The "exp" (expiration time) claim identifies the expiration time on or after which the token MUST NOT be accepted for processing. The processing of the "exp" claim requires that the current date/time MUST be before the expiration date/time listed in the "exp" claim. Implementers MAY provide for some small leeway, usually no more than a few minutes, to account for clock skew. This claim is OPTIONAL.
iss	string	StringAndURI	

			The "iss" (issuer) claim identifies the principal that issued the JWT. The processing of this claim is generally application specific. This claim is OPTIONAL.
aud	string	StringAndURI	The "aud" (audience) claim identifies the audience that the JWT is intended for. The processing of this claim requires that if a JWT consumer receives a JWT with an "aud" value that does not identify itself as the JWT audience, then the JWT MUST be rejected. The interpretation of the audience value is generally application specific. This claim is OPTIONAL.
typ	string	StringAndURI	The "typ" (type) claim is used to declare a type for the contents this JWT. The value MAY be a MIME (Freed, N. and N. Borenstein, "Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies," November 1996.) [RFC2045] type. This claim is OPTIONAL.

Table 1: Reserved Claim Definitions

Additional reserved claim names MAY be defined via the IANA JSON Web Token Claims registry, as per Section 9 (IANA Considerations). The syntaxes referred to above are:

Syntax Name	Syntax Definition
StringAndURI	Any string value MAY be used but a value containing a ":" character MUST be a URI as defined in RFC 3986 (Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax," January 2005.) [RFC3986].
URI	A URI as defined in RFC 3986 (Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax," January 2005.) [RFC3986].
IntDate	The number of seconds from 1970-01-01T0:0:0Z as measured in UTC until the desired date/time. See RFC 3339 (Klyne, G., Ed. and C. Newman, "Date and Time on the Internet: Timestamps," July 2002.) [RFC3339] for details regarding date/times in general and UTC in particular.

Table 2

4.2. Public Claim Names

TOC

Claim names can be defined at will by those using JWTs. However, in order to prevent collisions, any new claim name SHOULD either be defined in the IANA JSON Web Token Claims registry or be defined as a URI that contains a collision resistant namespace. Examples of collision resistant namespaces include:

- Domain Names,
- Object Identifiers (OIDs) as defined in the ITU-T X 660 and X 670 Recommendation series or
- Universally Unique IDentifier (UUID) as defined in RFC 4122 (Leach, P., Mealling, M., and R. Salz, "A Universally Unique IDentifier (UUID) URN Namespace," July 2005.) [RFC4122].

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 claim name.

4.3. Private Claim Names

TOC

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

5. JWT Envelope

TOC

The members of the JSON object represented by the Decoded JWT Envelope Segment describe the signature applied to the JWT Claim Segment and optionally additional properties of the JWT. Implementations MUST understand the entire contents of the envelope; otherwise, the JWT MUST be rejected for processing.

5.1. Reserved Envelope Parameter Names

TOC

The following envelope parameter names are reserved. All the names are short because a core goal of JWTs is for the tokens themselves to be short.

Envelope Parameter Name	JSON Value Type	Envelope Parameter Syntax	Envelope Parameter Semantics
alg	string	StringAndURI	The "alg" (algorithm) envelope parameter identifies the cryptographic algorithm used to secure the JWT. A list of reserved alg values is in Table 4 (JSON Web Token Reserved Algorithm Values). The processing of the "alg" (algorithm) envelope parameter, if present, requires that the value of the "alg" envelope parameter MUST be one that is both supported and for which there exists a key for use with that algorithm associated with the issuer of the JWT. Note however, that if the "iss" (issuer) claim is not included in the JWT Claim Segment, then the manner in which the issuer is determined is application specific. This envelope parameter is REQUIRED.
typ	string	StringAndURI	The "typ" (type) envelope parameter is used to declare that this data structure is a JWT. If a "typ" parameter is present, its value MUST be "JWT". This envelope parameter is OPTIONAL. (Non-normative note: Other values could be used by other specifications to declare data structures other than JWTs, for instance, encrypted JSON tokens.)
keyid	string	String	The "keyid" (key ID) envelope 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. Omitting this parameter is equivalent to setting it to an empty string. The format of this

			parameter is unspecified. This envelope parameter is OPTIONAL.
curi	string	URI	The "curi" (certificates URI) envelope parameter is a URI that points to X.509 public key certificates that can be used to validate the signature. This envelope parameter is OPTIONAL.
ctp	string	String	The "ctp" (certificate thumbprint) envelope parameter provides a base64url encoded SHA-1 thumbprint of the DER encoding of a certificate that can be used to validate the signature. This envelope parameter is OPTIONAL.

Table 3: Reserved Envelope Parameter Definitions

Additional reserved envelope parameter names MAY be defined via the IANA JSON Web Token Envelope Parameters registry, as per Section 9 (IANA Considerations). The envelope value syntaxes referred to above are defined in Table 2.

5.2. Public Envelope Parameter Names

TOC

Additional envelope parameter names can be defined by those using JWTs. However, in order to prevent collisions, any new envelope parameter name or algorithm value SHOULD either be defined in the IANA JSON Web Token Envelope 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 envelope parameter name.
New envelope parameters should be introduced sparingly, as they can result in non-interoperable JWTs. Nonetheless, some extensions needed for some use cases may require them, such as an extension to enable the inclusion of multiple signatures.

5.3. Private Envelope Parameter Names

TOC

A producer and consumer of a JWT may agree to any envelope parameter name that is not a Reserved Name Section 5.1 (Reserved Envelope Parameter Names) or a Public Name Section 5.2 (Public Envelope

Parameter Names). Unlike Public Names, these private names are subject to collision and should be used with caution.
New envelope parameters should be introduced sparingly, as they can result in non-interoperable JWTs.

---

6. General Rules for Creating and Validating a JWT

TOC

To create a JWT one MUST follow these steps:

1. Create a JSON object containing the desired claims. Note that white space is explicitly allowed in the representation and no canonicalization is performed before encoding.

2. Translate this JSON object's Unicode code points into UTF-8, as defined in RFC 3629 (Yergeau, F., “UTF-8, a transformation format of ISO 10646,” November 2003.) [RFC3629].

3. Base64url encode the UTF-8 representation of this JSON object as defined in this specification (without padding). This encoding becomes the JWT Claim Segment.

4. Create a different JSON object containing the desired envelope parameters. Note that white space is explicitly allowed in the representation and no canonicalization is performed before encoding.

5. Translate this JSON object's Unicode code points into UTF-8, as defined in RFC 3629 (Yergeau, F., “UTF-8, a transformation format of ISO 10646,” November 2003.) [RFC3629].

6. Base64url encode the UTF-8 representation of this JSON object as defined in this specification (without padding). This encoding becomes the JWT Envelope Segment.

7. Construct the JWT Crypto Segment as defined for the particular algorithm being used. The "alg" envelope parameter MUST be present in the JSON Envelope Segment, with the algorithm value accurately representing the algorithm used to construct the JWT Crypto Segment.

8. Combine the JWT Envelope Segment, the JWT Claim Segment and then the JWT Crypto Segment in that order, separating each by period characters, to create the JWT.

When validating a JWT the following steps MUST be taken. If any of the listed steps fails then the token MUST be rejected for processing.

1. The JWT MUST contain two period characters.

2. The JWT MUST be split on the two period characters resulting in three non-empty segments. The first segment is the JWT Envelope Segment; the second is the JWT Claim Segment; the third is the JWT Crypto Segment.

3. The JWT Envelope Segment MUST be successfully base64url decoded following the restriction given in this spec that no padding characters may have been used.

4. The Decoded JWT Envelope Segment MUST be completely valid JSON syntax.

5. The JWT Claim Segment MUST be successfully base64url decoded following the restriction given in this spec that no padding characters may have been used.

6. The Decoded JWT Claim Segment MUST be completely valid JSON syntax.

7. The JWT Crypto Segment MUST be successfully base64url decoded following the restriction given in this spec that no padding characters may have been used.

8. The JWT Envelope Segment MUST be validated to only include parameters and values whose syntax and semantics are both understood and supported.

9. When used in a security-related context, the JWT Claim Segment MUST be validated to only include claims whose syntax and semantics are both understood and supported.

10. The JWT Crypto Segment MUST be successfully validated against the JWT Claim Segment in the manner defined for the algorithm being used, which MUST be accurately represented by the value of the "alg" envelope parameter, which MUST be present.

Processing a JWT inevitably requires comparing known strings to values in the token. For example, in checking what the algorithm is, the Unicode string encoding "alg" will be checked against the member names in the Decoded JWT Envelope Segment to see if there is a matching envelope parameter name. A similar process occurs when determining if the value of the "alg" envelope parameter represents a supported algorithm. Comparing Unicode strings, however, has significant security implications, as per Section 10 (Security Considerations).
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 (Davis, M., Whistler, K., and M. Dürst, "Unicode Normalization Forms," 09 2009.) [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.

---

7. Base64url encoding as used by JWTs

TOC

JWTs make use of the base64url encoding as defined in RFC 4648 (Josefsson, S., "The Base16, Base32, and Base64 Data Encodings," October 2006.) [RFC4648]. As allowed by Section 3.2 of the RFC, this specification mandates that base64url encoding when used with JWTs MUST NOT use padding. The reason for this restriction is that the padding character ('=') is not URL safe.
For notes on implementing base64url encoding without padding, see Section 14 (Appendix - Non-Normative - Notes on implementing base64url encoding without padding).

---

8. Signing JWTs with Cryptographic Algorithms

TOC

JWTs use specific cryptographic algorithms to sign the contents of the JWT Claim Segment. The use of the following algorithms for producing JWTs is defined in this section. The table below is the list of "alg" envelope parameter values reserved by this specification, each of which is explained in more detail in the following sections:

Alg Claim 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 4: JSON Web Token Reserved Algorithm Values

Of these algorithms, only HMAC SHA-256 and RSA SHA-256 MUST be implemented. It is RECOMMENDED that implementations also implement at least the ECDSA P-256 SHA-256 algorithm.

8.1. Signing a JWT with HMAC SHA-256

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 JWT Claim Segment, which therefore demonstrates that whoever generated the MAC was in possession of the secret.
The algorithm for implementing and validating HMACs is provided in RFC 2104 (Krawczyk, H., Bellare, M., and R. Canetti, “HMAC: Keyed-Hashing for Message Authentication,” February 1997.) [RFC2104]. Although any HMAC can be used with JWTs, this section defines the use of the SHA-256 cryptographic hash function as defined in FIPS 180-3 (National Institute of Standards and Technology, “Secure Hash Standard (SHS),” October 2008.) [FIPS.180-3]. The reserved "alg" envelope parameter value "HS256" is used in the JWT Envelope Segment to indicate that the JWT Crypto Segment contains a base64url encoded HMAC SHA-256 HMAC value.
The HMAC SHA-256 MAC is generated as follows:

1. Take the bytes of the UTF-8 representation of the JWT Claim Segment and execute the HMAC SHA-256 algorithm on them using the shared key to produce an HMAC.

2. Base64url encode the HMAC as defined in this document.

The output is placed in the JWT Crypto Segment for that JWT.
The HMAC SHA-256 MAC on a JWT is validated as follows:

1. Take the bytes of the UTF-8 representation of the JWT Claim Segment and calculate an HMAC SHA-256 MAC on them using the shared key.

2. Base64url encode the previously generated HMAC as defined in this document.

3. If the JWT Crypto Segment and the previously calculated value exactly match in a character by character, case sensitive comparison, then one has confirmation that the key was used to generate the HMAC on the JWT and that the contents of the JWT Claim Segment have not be tampered with.

4. If the validation fails, the token 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.
JWT implementations MUST support the HMAC SHA-256 algorithm. Support for the HMAC SHA-384 and HMAC SHA-512 algorithms is OPTIONAL.

8.2. Signing a JWT with RSA SHA-256

TOC

This section defines the use of the RSASSA-PKCS1-v1_5 signature algorithm as defined in RFC 3447 (Jonsson, J. and B. Kaliski, “Public-Key Cryptography Standards (PKCS) #1: RSA Cryptography Specifications Version 2.1,” February 2003.) [RFC3447], Section 8.2 (commonly known as PKCS#1), using SHA-256 as the hash function. Note that the use of the RSASSA-PKCS1-v1_5 algorithm is permitted in FIPS 186-3 (National Institute of Standards and Technology, “Digital Signature Standard (DSS),” June 2009.) [FIPS.186-3], Section 5.5, as is the SHA-256 cryptographic hash function, which is defined in FIPS 180-3 (National Institute of Standards and Technology, “Secure Hash Standard (SHS),” October 2008.) [FIPS.180-3]. The reserved "alg" envelope parameter value "RS256" is used in the JWT Envelope Segment to indicate that the JWT Crypto Segment contains an RSA SHA-256 signature.
A 2048-bit or longer key length MUST be used with this algorithm.
The RSA SHA-256 signature is generated as follows:

1. Let K be the signer's RSA private key and let M be the bytes of the UTF-8 representation of the JWT Claim Segment.

2. Compute the octet string S = RSASSA-PKCS1-V1_5-SIGN (K, M).

3. Base64url encode the octet string S, as defined in this document.

The output is placed in the JWT Crypto Segment for that JWT.

The RSA SHA-256 signature on a JWT is validated as follows:

1. Take the JWT Crypto Segment and base64url decode it into an octet string S. If decoding fails, then the token MUST be rejected.

2. Let M be the bytes of the UTF-8 representation of the JWT Claim Segment and let (n, e) be the public key corresponding to the private key used by the signer.

3. Validate the signature with RSASSA-PKCS1-V1_5-VERIFY ((n, e), M, S).

4. If the validation fails, the token 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.
JWT implementations MUST support the RSA SHA-256 algorithm. Support for the RSA SHA-384 and RSA SHA-512 algorithms is OPTIONAL.

8.3. Signing a JWT with ECDSA P-256 SHA-256

TOC

The Elliptic Curve Digital Signature Algorithm (ECDSA) is defined by FIPS 186-3 (National Institute of Standards and Technology, “Digital Signature Standard (DSS),” June 2009.) [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. The P-256 curve is also defined in FIPS 186-3. The reserved "alg" envelope parameter value "ES256" is used in the JWT Envelope Segment to indicate that the JWT Crypto Segment contains a ECDSA P-256 SHA-256 signature.
A JWT is signed with an ECDSA P-256 SHA-256 signature as follows:

1. Take the bytes of the UTF-8 representation of the JWT Claim Segment and generate a digital signature of them 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 becomes the JWT Crypto Segment for the JWT.
The following procedure is used to validate the ECDSA signature of a JWT:

1. Take the JWT Crypto Segment and base64url decode it into a byte array. If decoding fails, the token 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 bytes of the UTF-8 representation of the JWT Claim Segment, R, S and the public key (x, y) to the ECDSA P-256 SHA-256 validator.

5. If the validation fails, the token 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.
It is RECOMMENDED that JWT implementations support the ECDSA P-256 SHA-256 algorithm. Support for the ECDSA P-384 SHA-384 and ECDSA P-521 SHA-512 algorithms is OPTIONAL.

8.4. Additional Algorithms

TOC

Additional algorithms MAY be used to protect JWTs with corresponding "alg" envelope parameter values being defined to refer to them. Like claim names, new "alg" envelope parameter values SHOULD either be defined in the IANA JSON Web Token Algorithms registry or be a URI that contains a collision resistant namespace. In particular, the use of algorithm identifiers defined in XML DSIG (Eastlake, D., Reagle, J., and D. Solo, “(Extensible Markup Language) XML-Signature Syntax and

Processing," March 2002.) [RFC3275] and related specifications is permitted.

---

9. IANA Considerations

TOC

This specification calls for:

- A new IANA registry entitled "JSON Web Token Claims" for reserved claim names Section 4.1 (Reserved Claim Names) used in a Decoded JWT Claim Segment. Inclusion in the registry is RFC Required in the RFC 5226 (Narten, T. and H. Alvestrand, "Guidelines for Writing an IANA Considerations Section in RFCs," May 2008.) [RFC5226] sense for reserved JWT claim names that are intended to be interoperable between implementations. The registry will just record the reserved claim name and a pointer to the RFC that defines it. This specification defines inclusion of the claim names defined in Table 1 (Reserved Claim Definitions).

- A new IANA registry entitled "JSON Web Token Envelope Parameters" for reserved envelope parameter names Section 5.1 (Reserved Envelope Parameter Names) used in a Decoded JWT Envelope Parameter Segment. Inclusion in the registry is RFC Required in the RFC 5226 (Narten, T. and H. Alvestrand, "Guidelines for Writing an IANA Considerations Section in RFCs," May 2008.) [RFC5226] sense for reserved JWT envelope parameter names that are intended to be interoperable between implementations. The registry will just record the reserved envelope parameter name and a pointer to the RFC that defines it. This specification defines inclusion of the envelope parameter names defined in Table 3 (Reserved Envelope Parameter Definitions).

- A new IANA registry entitled "JSON Web Token Algorithms" for reserved values used with the "alg" envelope parameter values used in a decoded JWT Envelope Segment. Inclusion in the registry is RFC Required in the RFC 5226 (Narten, T. and H. Alvestrand, "Guidelines for Writing an IANA Considerations Section in RFCs," May 2008.) [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 4 (JSON Web Token Reserved Algorithm Values).

TOC

10. 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 claim 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. Case sensitivity and more generally Unicode comparison issues that can cause security holes, especially in claim names and explain why Unicode Normalization is such a problem.
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. While in non-security contexts it's o.k. to be generous in what one accepts, in security contexts this can lead to serious security holes. For example, malformed JSON might indicate that someone has managed to find a security hole in the issuer's code and is leveraging it to get the issuer to issue "bad" tokens whose content the attacker can control.

10.1. Unicode Comparison Security Issues

TOC

Claim names in JWTs are Unicode strings. For security reasons, the representations these names must be compared verbatim after performing any escape processing (as per RFC 4627 (Crockford, D., “The application/json Media Type for JavaScript Object Notation (JSON),” July 2006.) [RFC4627], Section 2.5). In particular, Unicode Normalization (Davis, M., Whistler, K., and M. Dürst, “Unicode Normalization Forms,” 09 2009.) [USA15] or case folding MUST NOT be applied at any point to either the JSON string or to the string it is to be compared against.
This means, for instance, that these JSON strings must compare as being equal ("JWT", "\u004aWT"), whereas these must all compare as being not equal to the first set or to each other ("jwt", "Jwt", "JW\u0074").
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, JWT implementations SHOULD ensure that characters outside the Basic Multilingual Plane are preserved and compared correctly; alternatively, if this is not possible due to these characters exercising limitations present in the underlying JSON implementation, then input containing them MUST be rejected.

---

11. Open Issues

TOC

The following open issues have been identified during review of previous drafts. Additional input on them is solicited.

*The draft currently defines no mechanism(s) for retrieving public keys that are not encoded as X.509 certificates. A mechanism or mechanisms similar to the Magic Signatures key discovery process for Magic Keys could be added to future drafts. Some have suggested that they keys themselves also be encoded as JWTs.

*Related to the above, it's not clear whether the "iss" claim should be expected to contain a location for retrieving non-X.509 public keys, or whether a separate issuer key location parameter should be defined. Also, does this belong in the envelope or the claims?

---

12. Acknowledgements

TOC

The authors acknowledge that the design of JWTs was intentionally influenced by the design and simplicity of Simple Web Tokens (Hardt, D. and Y. Goland, “Simple Web Token (SWT),” November 2009.) [SWT]. Solutions for signing JSON tokens were also previously explored by Magic Signatures (Panzer (editor), J., Laurie, B., and D. Balfanz, “Magic Signatures,” August 2010.) [MagicSignatures], JSON Simple Sign (Bradley, J. and N. Sakimura (editor), “JSON Simple Sign,” September 2010.) [JSS], and Canvas Applications (Facebook, “Canvas Applications,” 2010.) [CanvasApp], all of which influenced this draft.

---

13. Appendix - Non-Normative - JWT Examples

TOC

13.1. JWT using HMAC SHA-256

TOC

TOC

13.1.1. Encoding

The Decoded JWT Claim Segment used in this example is:

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

Note that white space is explicitly allowed in Decoded JWT Claim Segments and no canonicalization is performed before encoding. The following byte array contains the UTF-8 characters for the Decoded JWT Claim Segment:
[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 JWT Claim Segment value:

```
eyJpc3MiOiJqb2UiLA0KICJleHAiOjEzMDA4MTkzODAsDQogImh0dHA6Ly9leGFtcGxlLmNvbS9pc19yb290Ijp0
```

The following example JSON envelope object declares that the encoded object is a JSON Web Token (JWT) and the JWT Claim Segment is signed using the HMAC SHA-256 algorithm:

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

The following byte array contains the UTF-8 characters for the Decoded JWT Envelope Segment:
[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 JWT Envelope Segment value:

```
eyJ0eXAiOiJKV1QiLA0KICJhbGciOiJIUzI1NiJ9
```

HMACs are generated using keys. This example used the key represented by the following byte array:
[83, 159, 117, 12, 235, 169, 168, 200, 131, 152, 227, 246, 214, 212,
188, 74, 71, 83, 244, 166, 90, 24, 239, 251, 32, 124, 6, 201, 194, 104,
241, 62, 174, 246, 65, 111, 49, 52, 210, 118, 212, 124, 34, 88, 167,
112, 84, 88, 83, 65, 155, 18, 234, 250, 224, 101, 147, 221, 23, 104,
219, 170, 146, 215]
Running the HMAC SHA-256 algorithm on the JWT Claim Segment with this key yields the following byte array:
[223, 155, 172, 90, 63, 87, 240, 124, 6, 75, 224, 131, 115, 29, 73, 63,
99, 102, 169, 202, 203, 193, 158, 4, 42, 159, 44, 53, 56, 95, 221, 198]
Base64url encoding the above HMAC output yields the JWT Crypto Segment value:

```
35usWj9X8HwGS-CDcx1JP2NmqcrLwZ4EKp8sNThf3cY
```

Combining these segments in the order Envelope.Claims.Signature with period characters between the segments yields this complete JWT (with line breaks for display purposes only):

```
eyJ0eXAiOiJKV1QiLA0KICJhbGciOiJIUzI1NiJ9
.
eyJpc3MiOiJqb2UiLA0KICJleHAiOjEzMDA4MTkzODAsDQogImh0dHA6Ly9leGFtcGxlLmNvbS9pc19yb290IjpU
.
35usWj9X8HwGS-CDcx1JP2NmqcrLwZ4EKp8sNThf3cY
```

13.1.2. Decoding

TOC

Decoding the JWT first requires removing the base64url encoding from the JWT Envelope Segment and the JWT Claim Segment. We base64url decode the segments per Section 7 (Base64url encoding as used by JWTs) and turn them into the corresponding UTF-8 byte arrays, which we then translate into the Decoded JWT Envelope Segment and Decoded JWT Claim Segment strings.

13.1.3. Validating

TOC

Next we validate the decoded results. Since the "alg" parameter in the envelope is "HS256", we validate the HMAC SHA-256 signature contained in the JWT Crypto Segment. If any of the validation steps fail, the token MUST be rejected.
First, we validate that the decoded envelope and claim segment strings are both legal JSON.
To validate the signature, we repeat the previous process of using the correct key and the JWT Claim Segment as input to a SHA-256 HMAC function and then taking the output, base64url encoding it, and determining if it matches the JWT Crypto Segment in the JWT. If it matches exactly, the token has been validated.

13.2. JWT using RSA SHA-256

TOC

13.2.1. Encoding

TOC

The Decoded JWT Claim Segment used in this example is the same as in the previous example:

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

Since the JWT Claim Segment will therefore be the same, its computation is not repeated here. However, the Decoded JWT Envelope Segment is different 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 Decoded JWT Envelope Segment used is:

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

The following byte array contains the UTF-8 characters for the Decoded JWT Envelope Segment:
[123, 34, 97, 108, 103, 34, 58, 34, 82, 83, 50, 53, 54, 34, 125]
Base64url encoding this UTF-8 representation yields this JWT Envelope Segment value:

```
eyJhbGciOiJSUzI1NiJ9
```

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	[210, 252, 123, 106, 10, 30, 108, 103, 16, 74, 235, 143, 136, 178, 87, 102, 155, 77, 246, 121, 221, 173, 9, 155, 92, 74, 108, 217, 168, 128, 21, 181, 161, 51, 191, 11, 133, 108, 120, 113, 182, 223, 0, 11, 85, 79, 206, 179, 194, 237, 81, 43, 182, 143, 20, 92, 110, 132, 52, 117, 47, 171, 82, 161, 207, 193, 36, 64, 143, 121, 181, 138, 69, 120, 193, 100, 40, 133, 87, 137, 247, 162, 73, 227, 132, 203, 45, 159, 174, 45, 103, 253, 150, 251, 146, 108, 25, 142, 7, 115, 153, 253, 200, 21, 192, 175, 9, 125, 222, 90, 173, 239, 244, 77, 231, 14, 130, 127, 72, 120, 67, 36, 57, 191, 238, 185, 96, 104, 208, 71, 79, 197, 13, 109, 144, 191, 58, 152, 223, 175, 16, 64, 200, 156, 2, 214, 146, 171, 59, 60, 40, 150, 96, 157, 134, 253, 115, 183, 116, 206, 7, 64, 100, 124, 238, 234, 163, 16, 189, 18, 249, 133, 168, 235, 159, 89, 253, 212, 38, 206, 165, 178, 18, 15, 79, 42, 52, 188, 171, 118, 75, 126, 108, 84, 214,

	132, 2, 56, 188, 196, 5, 135, 165, 158, 102, 237, 31, 51, 137, 69, 119, 99, 92, 71, 10, 247, 92, 249, 44, 32, 209, 218, 67, 225, 191, 196, 25, 226, 34, 166, 240, 208, 187, 53, 140, 94, 56, 249, 203, 5, 10, 234, 254, 144, 72, 20, 241, 172, 26, 164, 156, 202, 158, 160, 202, 131]
e	[1, 0, 1]
d	[95, 135, 19, 181, 226, 88, 254, 9, 248, 21, 131, 236, 92, 31, 43, 117, 120, 177, 230, 252, 44, 131, 81, 75, 55, 145, 55, 17, 161, 186, 68, 154, 21, 31, 225, 203, 44, 160, 253, 51, 183, 113, 230, 138, 59, 25, 68, 100, 157, 200, 103, 173, 28, 30, 82, 64, 187, 133, 62, 95, 36, 179, 52, 89, 177, 64, 40, 210, 214, 99, 107, 239, 236, 30, 141, 169, 116, 179, 82, 252, 83, 211, 246, 18, 126, 168, 163, 194, 157, 209, 79, 57, 65, 104, 44, 86, 167, 135, 104, 22, 78, 77, 218, 143, 6, 203, 249, 199, 52, 170, 232, 0, 50, 36, 39, 142, 169, 69, 74, 33, 177, 124, 176, 109, 23, 128, 117, 134, 140, 192, 91, 61, 182, 255, 29, 253, 195, 213, 99, 120, 180, 237, 173, 237, 240, 195, 122, 76, 220, 38, 209, 212, 154, 194, 111, 111, 227, 181, 34, 10, 93, 210, 147, 150, 98, 27, 188, 104, 140, 242, 238, 226, 198, 224, 213, 77, 163, 199, 130, 1, 76, 208, 115, 157, 178, 82, 204, 81, 202, 235, 168, 211, 241, 184, 36, 186, 171, 36, 208, 104, 236, 144, 50, 100, 215, 214, 120, 171, 8, 240, 110, 201, 231, 226, 61, 150, 6, 40, 183, 68, 191, 148, 179, 105, 70, 86, 70, 60, 126, 65, 115, 153, 237, 115, 208, 118, 200, 145, 252, 244, 99, 169, 170, 156, 230, 45, 169, 205, 23, 226, 55, 220, 42, 128, 2, 241]

The RSA private key (n, d) is then passed to the RSA signing function, which also takes the hash type, SHA-256, and the JWT Claim Segment 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	[208, 141, 219, 44, 66, 129, 179, 230, 69, 120, 123, 108, 203, 96, 182, 145, 66, 179, 198, 104, 43, 187, 199, 159, 175, 5, 217, 101, 109, 236, 88, 136, 193, 133, 79, 39, 162, 131, 58, 114, 133, 202, 171, 227, 135, 157, 123, 188, 90, 111, 66, 241, 38, 238, 59, 18, 125, 146, 129, 14, 54, 183, 10, 221, 33, 105, 37, 173, 119, 239, 92, 27, 232, 175, 173, 49, 21, 28, 252, 237, 183, 107, 98, 156, 113, 116, 162, 219, 53, 96, 44, 214, 175, 154, 61, 100, 175, 90, 118, 247, 42, 196, 45, 74, 217, 145, 92, 39, 123, 224, 247, 171, 206, 203, 91, 167, 103, 57, 163, 87, 172, 67, 77, 255, 9, 218, 107, 62, 228, 71, 239, 36, 246, 23, 96, 108, 28, 19, 179, 24, 167, 196, 42, 97, 198, 80, 241, 79, 31, 0, 85, 17, 50, 6, 143, 238, 214, 131, 246, 13, 49, 111, 30, 142, 182, 145, 200, 17, 127, 76, 236, 69, 66, 133, 198, 137, 103, 45, 3, 48, 123, 203, 17, 162, 1, 105, 133, 22, 105, 25, 63, 173, 186, 231,

```
206, 246, 22, 243, 250, 53, 237, 209, 36, 111, 168, 11, 40, 237,
179, 83, 125, 180, 84, 231, 129, 37, 236, 172, 22, 234, 58, 198,
187, 124, 65, 145, 148, 227, 122, 177, 16, 176, 84, 28, 1, 141,
179, 57, 96, 232, 215, 51, 7, 49, 63, 195, 155, 94, 51, 22, 239,
90, 138, 207, 41, 62]
```

Base64url encoding the signature produces this value for the JWT Crypto Segment:

```
0I3bLEKBs-ZFeHtsy2C2kUKzxmgru8efrwXZZW3sWIjBhU8nooM6coXKq-OHnXu8Wm9C8SbuOxJ9koEONrcK3SFp
```

Combining these segments in the order Envelope.Claims.Signature with period characters between the segments yields this complete JWT (with line breaks for display purposes only):

```
eyJhbGciOiJSUzI1NiJ9
.
eyJpc3MiOiJqb2UiLA0KICJleHAiOjEzMDA4MTkzODAsDQogImh0dHA6Ly9leGFtcGxlLmNvbS9pc19yb290IjpC
.
0I3bLEKBs-ZFeHtsy2C2kUKzxmgru8efrwXZZW3sWIjBhU8nooM6coXKq-OHnXu8Wm9C8SbuOxJ9koEONrcK3SFp
```

13.2.2. Decoding

TOC

Decoding the JWT from this example requires processing the JWT Envelope Segment and Claim Segment exactly as done in the first example.

13.2.3. Validating

TOC

Since the "alg" parameter in the envelope is "RS256", we validate the RSA SHA-256 signature contained in the JWT Crypto Segment. If any of the validation steps fail, the token MUST be rejected.
First, we validate that the decoded envelope and claim segment strings are both legal JSON.
Validating the JWT Crypto Segment is a little different from the previous example. First, we base64url decode the JWT Crypto Segment to produce a signature S to check. We then pass (n, e), S and the JWT Claim Segment to an RSA signature verifier that has been configured to use the SHA-256 hash function.

TOC

13.3. JWT using ECDSA P-256 SHA-256

13.3.1. Encoding

TOC

The Decoded JWT Claim Segment used in this example is the same as in the previous examples:

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

Since the JWT Claim Segment will therefore be the same, its computation is not repeated here. However, the Decoded JWT Envelope Segment is differs from the previous example because a different algorithm is being used. The Decoded JWT Envelope Segment used is:

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

The following byte array contains the UTF-8 characters for the Decoded JWT Envelope Segment:
[123, 34, 97, 108, 103, 34, 58, 34, 69, 83, 50, 53, 54, 34, 125]
Base64url encoding this UTF-8 representation yields this JWT Envelope Segment value:

```
eyJhbGciOiJFUzI1NiJ9
```

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	[48, 160, 66, 76, 210, 28, 41, 68, 131, 138, 45, 117, 201, 43, 55, 231, 110, 162, 13, 159, 0, 137, 58, 59, 78, 238, 138, 60, 10, 175, 236, 62]
y	[224, 75, 101, 233, 36, 86, 217, 136, 139, 82, 179, 121, 189, 251, 213, 30, 232, 105, 239, 31, 15, 198, 91, 102, 89, 105, 91, 108, 206, 8, 23, 35]
d	[243, 189, 12, 7, 168, 31, 185, 50, 120, 30, 213, 39, 82, 246, 12, 200, 154, 107, 229, 229, 25, 52, 254, 1, 147, 141, 219, 85, 216, 247, 120, 1]

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 JWT Claim Segment 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	[175, 11, 115, 42, 160, 182, 181, 28, 135, 222, 52, 154, 182, 237, 206, 137, 82, 20, 243, 7, 12, 164, 107, 72, 236, 187, 241, 190, 26, 76, 32, 181]
S	[120, 23, 189, 205, 202, 13, 177, 187, 23, 47, 12, 227, 237, 250, 230, 233, 245, 216, 9, 170, 24, 185, 198, 187, 193, 94, 158, 117, 167, 88, 153, 196]

Concatenating the S array to the end of the R array and base64url encoding the result produces this value for the JWT Crypto Segment:

```
rwtzKqC2tRyH3jSatu3OiVIU8wcMpGtI7LvxvhpMILV4F73Nyg2xuxcvDOPt-ubp9dgJqhi5xrvBXp51p1iZxA
```

Combining these segments in the order Envelope.Claims.Signature with period characters between the segments yields this complete JWT (with line breaks for display purposes only):

```
eyJhbGciOiJFUzI1NiJ9
.
eyJpc3MiOiJqb2UiLA0KICJleHAiOjEzMDA4MTkzODAsDQogImh0dHA6Ly9leGFtcGxlLmNvbS9pc19yb290IjpD
.
rwtzKqC2tRyH3jSatu3OiVIU8wcMpGtI7LvxvhpMILV4F73Nyg2xuxcvDOPt-ubp9dgJqhi5xrvBXp51p1iZxA
```

13.3.2. Decoding

TOC

Decoding the JWT from this example requires processing the JWT Envelope Segment and Claim Segment exactly as done in the first example.

13.3.3. Validating

TOC

Since the "alg" parameter in the envelope is "ES256", we validate the ECDSA P-256 SHA-256 signature contained in the JWT Crypto Segment. If any of the validation steps fail, the token MUST be rejected.
First, we validate that the decoded envelope and claim segment strings are both legal JSON.
Validating the JWT Crypto Segment is a little different from the first example. First, we base64url decode the JWT Crypto Segment 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 JWT Claim Segment 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 8.3 (Signing a JWT with ECDSA P-256 SHA-256), 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.

---

14. Appendix - Non-Normative - Notes on implementing base64url encoding without padding

TOC

This appendix describes how to implement base64url encoding and decoding functions without padding based upon standard base64 encoding and decoding functions that do use padding.
To be concrete, example C# code implementing these functions is shown below. Similar code could be used in other languages.

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

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

As per the example code above, the number of '=' padding characters that needs to be added to the end of a base64url encoded string without

padding to turn it into one with padding is a deterministic function of the length of the encoded string. Specifically, if the length mod 4 is 0, no padding is added; if the length mod 4 is 2, two '=' padding characters are added; if the length mod 4 is 3, one '=' padding character is added; if the length mod 4 is 1, the input is malformed. An example correspondence between unencoded and encoded values follows. The byte sequence below encodes into the string below, which when decoded, reproduces the byte sequence.

```
3 236 255 224 193


A-z_4ME
```

---

15. Appendix - Non-Normative - Relationship of JWTs to SAML Tokens

TOC

SAML 2.0 (Cantor, S., Kemp, J., Philpott, R., and E. Maler, “Assertions and Protocol for the OASIS Security Assertion Markup Language (SAML) V2.0,” March 2005.) [OASIS.saml-core-2.0-os] provides a standard for creating tokens with much greater expressivity and more security options than supported by JWTs. However, the cost of this flexibility and expressiveness is both size and complexity. In addition, SAML's use of XML (Cowan, J., “Extensible Markup Language (XML) 1.1,” October 2002.) [W3C.CR-xml11-20021015] and XML DSIG (Eastlake, D., Reagle, J., and D. Solo, “(Extensible Markup Language) XML-Signature Syntax and Processing,” March 2002.) [RFC3275] only contributes to the size of SAML tokens.

JWTs are intended to provide a simple token format that is small enough to fit into HTTP headers and query arguments in URIs. It does this by supporting a much simpler token model than SAML and using the JSON (Crockford, D., “The application/json Media Type for JavaScript Object Notation (JSON),” July 2006.) [RFC4627] object encoding syntax. It also supports securing tokens using Hash-based Message Authentication Codes (HMACs) and digital signatures using a smaller (and less flexible) format than XML DSIG.

Therefore, while JWTs can do some of the things SAML tokens do, JWTs are not intended as a full replacement for SAML tokens, but rather as a compromise token format to be used when space is at a premium.

TOC

16. Appendix - Non-Normative - Relationship of JWTs to Simple Web Tokens (SWTs)

Both JWTs and Simple Web Tokens SWT (Hardt, D. and Y. Goland, "Simple Web Token (SWT)," November 2009.) [SWT], at their core, enable sets of claims to be communicated between applications. For SWTs, both the claim names and claim values are strings. For JWTs, while claim names are strings, claim values can be any JSON type. Both token types offer cryptographic protection of their content: SWTs with HMAC SHA-256 and JWTs with a choice of algorithms, including HMAC SHA-256, RSA SHA-256, and ECDSA P-256 SHA-256.

---

17. References

TOC

17.1. Normative References

TOC

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

17.2. Informative References

TOC

[CanvasApp]	Facebook, “Canvas Applications,” 2010.
[JSS]	Bradley, J. and N. Sakimura (editor), “JSON Simple Sign,” September 2010.
[MagicSignatures]	Panzer (editor), J., Laurie, B., and D. Balfanz, “Magic Signatures,” August 2010.
[OASIS.saml-core-2.0-os]	Cantor, S., Kemp, J., Philpott, R., and E. Maler, “Assertions and Protocol for the OASIS Security Assertion Markup Language (SAML) V2.0,” OASIS Standard saml-core-2.0-os, March 2005.
[RFC3275]	Eastlake, D., Reagle, J., and D. Solo, “(Extensible Markup Language) XML-Signature Syntax and Processing,” RFC 3275, March 2002 (TXT).
[RFC4122]	Leach, P., Mealling, M., and R. Salz, “A Universally Unique IDentifier (UUID) URN Namespace,” RFC 4122, July 2005 (TXT, HTML, XML).
[SWT]	Hardt, D. and Y. Goland, “Simple Web Token (SWT),” Version 0.9.5.1, November 2009.
[W3C.CR-xml11-20021015]	Cowan, J., “Extensible Markup Language (XML) 1.1,” W3C CR CR-xml11-20021015, October 2002.

Authors' Addresses

TOC

	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