Network Working Group	M. Jones	
Internet-Draft	Microsoft	
Intended status: Standards Track	D. Balfanz	
Expires: July 8, 2011	Google	
	J. Bradley	
	independent	
	Y. Goland	
	Microsoft	
	J. Panzer	
	Google	
	N. Sakimura	
	Nomura Research Institute	
	P. Tarjan	
	Facebook	
	January 04, 2011	

TOC

JSON Web Token (JWT) - Claims and Signing
draft-jones-json-web-token-01

Abstract

JSON Web Token (JWT) is a means of representing signed content using JSON data structures, including claims to be transferred between two parties. The claims in a JWT are encoded as a JSON object that is digitally signed and optionally encrypted. Encryption for JWTs is described in a separate companion specification.
The suggested pronunciation of JWT is the same as the English word "jot".

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

Copyright Notice

Table of Contents

---

1. Introduction

TOC

JSON Web Token (JWT) is a compact token format intended for space constrained environments such as HTTP Authorization headers and URI query parameters. JWTs encode 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 JWT signature mechanisms are independent of the type of content being signed,

allowing arbitrary content to be signed. Encryption for JWTs is described in a separate companion specification.
The suggested pronunciation of JWT is the same as the English word "jot".

---

2. Terminology

TOC

JSON Web Token (JWT) A data structure containing three JWT Token Segments: the JWT Header Segment, the JWT Payload Segment, and the JWT Crypto Segment. The JWT Payload Segment typically represents a set of claims convened by the JWT as a JSON object, but in the general case, may represent arbitrary signed content.

JWT Compact Serialization A data structure representing a JWT as a string consisting of three JWT Token Segments: the JWT Header Segment, the JWT Payload Segment, and the JWT Crypto Segment, in that order, with the segments being separated by period ('.') characters.

JWT JSON Serialization A data structure representing a JWT as a JSON object with members for each of three kinds of JWT Token Segments: a "header" member whose value is a non-empty array of JWT Header Segments, a "payload" member whose value is the JWT Payload Segment, and a "signature" member whose value is a non-

empty array of JWT Crypto Segments, where the cardinality of both arrays is the same.

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 Header Segment A JWT Token Segment containing a base64url encoded JSON object that describes the signature applied to the JWT Header Segment and the JWT Payload Segment.

JWT Payload Segment A JWT Token Segment containing base64url encoded content. This may be a JWT Claims Object.

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

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

Decoded JWT Payload Segment A JWT Payload Segment that has been base64url decoded. If the corresponding JWT Payload Segment is a JWT Claims Object, this will be a Decoded JWT Claims Object.

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

JWT Claims Object A base64url encoded JSON object that represents the claims contained in the JWT.

Decoded JWT Claims Object A JSON object that represents the claims contained in the JWT.

JWT Signing Input The concatenation of the JWT Header Segment, a period ('.') character, and the JWT Payload Segment.

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 (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.

Header Parameter Names The names of the members within the JSON object represented in a JWT Header Segment.

Header Parameter Values The values of the members within the JSON object represented in a JWT Header Segment.

Claim Names The names of the members of the JSON object represented in a JWT Claims Object.

Claim Values The values of the members of the JSON object represented in a JWT Claims Object.

---

3. JSON Web Token (JWT) Overview

TOC

JWTs represent content that is base64url encoded and digitally signed, and optionally encrypted, using JSON data structures; this content is typically a set of claims represented as a JSON object.
When the JWT payload is a set of claims, the claims are represented as name/value pairs that are members of a JSON object. The JSON object is base64url encoded to produce the JWT Claims Object, which is used as the JWT Payload Segment. An accompanying base64url encoded JSON header - the JWT Header Segment - describes the signature method used.
The names within the header object MUST be unique. The names within the header object are referred to as Header Parameter Names. The corresponding values are referred to as Header Parameter Values.
Likewise, if the payload represents a JWT Claims Object, the names within the claims object MUST be unique. The names within the claims 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 JWT Header Segment and the JWT Payload Segment. This signature value is carried in the JWT Crypto Segment. The JSON Header object MUST contain an "alg" parameter, the value of which is a string that unambiguously identifies the algorithm used to sign the JWT Header Segment and the JWT Payload 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 Claims Object:

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

Base64url encoding the UTF-8 representation of the JSON object yields this JWT Claims Object, which is used as the JWT Payload Segment:

```
eyJpc3MiOiJqb2UiLA0KICJleHAiOjEzMDA4MTkzODAsDQogImh0dHA6Ly9leGFtcGxlLmNvbS9pc19yb290Ijp0
```

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

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

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

```
eyJ0eXAiOiJKV1QiLA0KICJhbGciOiJIUzI1NiJ9
```

Signing the UTF-8 representation of the JWT Signing Input (the concatenation of the JWT Header Segment, a period ('.') character, and the JWT Payload 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:

```
dBjftJeZ4CVP-mB92K27uhbUJU1p1r_wW1gFWFOEjXk
```

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

```
eyJ0eXAiOiJKV1QiLA0KICJhbGciOiJIUzI1NiJ9
.
eyJpc3MiOiJqb2UiLA0KICJleHAiOjEzMDA4MTkzODAsDQogImh0dHA6Ly9leGFtcGxlLmNvbS9pc19yb290Ijp0
.
dBjftJeZ4CVP-mB92K27uhbUJU1p1r_wW1gFWFOEjXk
```

This computation is illustrated in more detail in Appendix A.1 (JWT using HMAC SHA-256).

TOC

4. JWT Claims

If the JWT contains a set of claims represented as a JSON object, then the members of the JSON object represented by the Decoded JWT Claims Object decoded from the JWT Payload 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. When used in a security-related context, implementations MUST understand and support all of the claims present; otherwise, the JWT MUST be rejected for processing.
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 principal intended to process the JWT MUST be identified by the value of the audience

			claim. If the principal processing the claim does not identify itself with the identifier in the "aud" claim value then the JWT MUST be rejected. The interpretation of the contents of the audience value is generally application specific. This claim is OPTIONAL.
typ	string	String	The "typ" (type) claim is used to declare a type for the contents of this JWT. 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 10 (IANA Considerations). The syntax values used above and in Table 3 (Reserved Header Parameter Definitions) 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 (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.
String	Any string value MAY be used.
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].
URL	A URL as defined in RFC 1738 (Berners-Lee, T., Masinter, L., and M. McCahill, “Uniform Resource Locators (URL),” December 1994.) [RFC1738].

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 Header

TOC

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

5.1. Reserved Header Parameter Names

TOC

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

Header Parameter Name	JSON Value Type	Header Parameter Syntax	Header Parameter Semantics
alg	string	StringAndURI	The "alg" (algorithm) header 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) 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 issuer of the JWT. This header parameter is REQUIRED.
typ	string	String	The "typ" (type) header parameter is used to declare that this data structure is a JWT. If a "typ" parameter is present, it is RECOMMENDED that its value be "JWT". This header parameter is OPTIONAL.
jku	string	URL	The "jku" (JSON Key URL) header parameter is a URL that points to JSON-encoded public key certificates that can be used to validate the signature. The specification for this encoding is TBD. 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. Omitting this parameter is equivalent to setting it to an empty string. 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 a URL that points to an X.509 public key certificate that can be used to validate the signature. This certificate MUST conform to RFC 5280 (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,” May 2008.) [RFC5280]. This header parameter is OPTIONAL.
x5t	string	String	The "x5t" (x.509 certificate thumbprint) header parameter provides a base64url encoded SHA-256 thumbprint (a.k.a. digest) of the DER encoding of an X.509 certificate that can be used to match a certificate. This header parameter is OPTIONAL.

Table 3: Reserved Header Parameter Definitions

Additional reserved header parameter names MAY be defined via the IANA JSON Web Token Header Parameters registry, as per Section 10 (IANA Considerations). The syntax values used above and in Table 1 (Reserved Claim Definitions) are defined in Table 2.

5.2. Public Header Parameter Names

TOC

Additional header parameter names can be defined by those using JWTs. However, in order to prevent collisions, any new header parameter name or algorithm value SHOULD either be defined in the IANA JSON Web Token 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 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 Header Parameter Names

TOC

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

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 JWTs.

---

6. Rules for Creating and Validating a JWT

TOC

To create a JWT one MUST follow these steps:

1. Create the payload content to be encoded as the Decoded JWT Payload Segment. If the payload represents a JWT Claims Object, then these steps for creating the Decoded JWT Payload Segment also apply:

   *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.

   *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]. This is the Decoded JWT Payload Segment.

2. Base64url encode the Decoded JWT Payload Segment. This encoding becomes the JWT Payload Segment.

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 (Yergeau, F., “UTF-8, a transformation format of ISO 10646,” November 2003.) [RFC3629].

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

6. Construct a JWT Crypto Segment as defined for the particular algorithm being used. The JWT Signing Input is always the concatenation of a JWT Header Segment, a period ('.') character, and the JWT Payload Segment. The "alg" header parameter MUST be present in the JSON Header Segment, with the algorithm value accurately representing the algorithm used to construct the JWT Crypto Segment.

7. If the JWT Compact Serialization is being used, then:

   * Concatenate the JWT Header Segment, the JWT Payload Segment and then the JWT Crypto Segment in that order, separating each by period characters, to create the JWT.

   Else if the JWT JSON Serialization is being used, then:

   * Create a JSON object with these three members: a "header" member whose value is an array of JWT Header Segments, a "payload" member whose value is the JWT Payload Segment, and a "signature" member whose value is an array of JWT Crypto Segments.

   * If more than one signature is present, then repeat steps 3 through 6 for each header and crypto segment to produce additional values for the header and signature arrays.

   * The header and signature arrays must have the same number of values, with each header value and corresponding signature value being located at the same array index.

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. If the JWT Compact Serialization is being used, then:

   * The JWT MUST contain two period characters.

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

   Else if the JWT JSON Serialization is being used, then:

   * The JSON MUST contain the three members "header", "payload", and "signature" and MAY contain others, which MUST be ignored. The payload member MUST be a string and the header and signature members MUST be non-empty arrays of strings with equal cardinality.

   * Use a "header" member array value as the JWT Header Segment; use the "payload" member value as the JWT Payload Segment; use a "signature" member array value with the same index as the "header" member array value used as the JWT Crypto Segment.

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

3. If the payload represents a JWT Claims Object, then these steps for validating the Decoded JWT Payload Segment also apply:

   *The Decoded JWT Payload Segment, which is the Decoded JWT Claims Object, MUST be completely valid JSON syntax conforming to RFC 4627 (Crockford, D., "The application/json Media Type for JavaScript Object Notation (JSON)," July 2006.) [RFC4627].

   *When used in a security-related context, the Decoded JWT Claims Object MUST be validated to only include claims whose syntax and semantics are both understood and supported.

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

5. The Decoded JWT Header Segment MUST be completely valid JSON syntax conforming to RFC 4627 (Crockford, D., "The application/json Media Type for JavaScript Object Notation (JSON)," July 2006.) [RFC4627].

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

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

8. The JWT Crypto Segment MUST be successfully validated against the JWT Header Segment and JWT Payload Segment 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.

9. If the JWT JSON Serialization is being used, then repeat steps 4 to 8 for each element of the header and signature arrays.

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 Header Segment 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 11 (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 Appendix B (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 Header Segment and the JWT Payload Segment. The use of the following algorithms for producing JWTs 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 4: JSON Web Token Reserved Algorithm Values

Of these algorithms, only HMAC SHA-256 and RSA SHA-256 MUST be implemented by conforming implementations. It is RECOMMENDED that implementations also support the ECDSA P-256 SHA-256 algorithm. Support for other algorithms is OPTIONAL.
The portion of a JWT that is signed is the same for all algorithms: the concatenation of the JWT Header Segment, a period ('.') character, and the JWT Payload Segment. This character sequence is referred to as the JWT Signing Input. Note that in the JWT Compact Serialization, this corresponds to the portion of the JWT representation preceding the second period character. The UTF-8 representation of the JWT Signing Input is passed to the respective signing algorithms.

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 Signing Input, 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" header parameter value "HS256" is used in the JWT Header 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. Apply the HMAC SHA-256 algorithm to the UTF-8 representation of the JWT Signing Input 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. Apply the HMAC SHA-256 algorithm to the UTF-8 representation of the JWT Signing Input of the JWT 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, then one has confirmation that the key was used to generate the HMAC on the JWT and that the contents of the JWT 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.

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 described 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" header parameter value "RS256" is used in the JWT Header 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 UTF-8 representation of the JWT Signing Input.

2. Compute the octet string S = RSASSA-PKCS1-V1_5-SIGN (K, M) using SHA-256 as the hash function.

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 UTF-8 representation of the JWT Signing Input 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) using SHA-256 as the hash function.

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.

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" header parameter value "ES256" is used in the JWT Header Segment to indicate that the JWT Crypto Segment contains an ECDSA P-256 SHA-256 signature.
A JWT is signed with an ECDSA P-256 SHA-256 signature as follows:

1. Generate a digital signature of the UTF-8 representation of the JWT 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 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 UTF-8 representation of the JWT Signing Input, 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.

8.4. Additional Algorithms

TOC

Additional algorithms MAY be used to protect JWTs with corresponding "alg" header parameter values being defined to refer to them. Like claim names, new "alg" header 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. JWT Serialization Formats

TOC

JSON Web Tokens (JWTs) support two serialization formats: the JWT Compact Serialization, which is more space efficient and intended for uses where the token is passed as a simple string-valued parameter, and the JWT JSON Serialization, which is more general, being able to contain multiple signatures over the same content. The two serialization formats are intended for use in different contexts.

9.1. JWT Compact Serialization

TOC

The JWT Compact Serialization represents a JWT as a string consisting of three JWT Token Segments: the JWT Header Segment, the JWT Payload Segment, and the JWT Crypto Segment, in that order, with the segments being separated by period ('.') characters. It is intended for uses where the token is passed as a simple string-valued parameter, including in URLs.

The Compact Serialization contains only one signature to keep this format simple. The example JWT in Section 3.1 (Example JWT) uses the Compact Serialization.

9.2. JWT JSON Serialization

TOC

The JWT JSON Serialization represents a JWT as a JSON object with members for each of three kinds of JWT Token Segments: a "header" member whose value is a non-empty array of JWT Header Segments, a "payload" member whose value is the JWT Payload Segment, and a "signature" member whose value is a non-empty array of JWT Crypto Segments, where the cardinality of both arrays is the same.

Unlike the Compact Serialization, JWTs using the JSON Serialization MAY contain multiple signatures. Each signature is represented as a JWT Crypto Segment in the "signature" member array. For each signature, there is a corresponding "header" member array element that specifies the signature algorithm for that signature, and potentially other information as well. Therefore, the syntax is:

```
{"header":["<header 1 contents>",...,"<header N contents>"],
 "payload":"<payload contents>",
 "signature":["<signature 1 contents>",...,"<signature N contents>"]
}
```

The i'th signature is computed on the concatenation of <header i contents>.<payload contents>.
Appendix A.4 (JWT using JSON Serialization) contains an example JWT using the JSON Serialization.

---

10. IANA Considerations

TOC

This specification calls for:

- A new IANA registry entitled "JSON Web Token Claims" for reserved claim names is defined in Section 4.1 (Reserved Claim Names). 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 Header Parameters" for reserved header parameter names is defined in Section 5.1 (Reserved Header Parameter Names). 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 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 3 (Reserved Header Parameter Definitions).

- A new IANA registry entitled "JSON Web Token Algorithms" for reserved values used with the "alg" header parameter values is defined in Section 8.4 (Additional Algorithms). 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).

11. Security Considerations

TOC

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.
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.

11.1. Unicode Comparison Security Issues

TOC

Claim names in JWTs are Unicode strings. For security reasons, the representations of 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).
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.

TOC

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

The following items remain to be done in this draft (and related drafts):

* The specification will be a lot clearer if the signature portions are cleanly separated from the claims token format and serialization portions. Having tried it this way and being dissatisfied with the sometimes unwieldy readability of the result, I plan to perform the separation in the next 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 claim names and header parameters 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 endpoints' private parameters.

* Clarify the optional ability to provide type information JWTs and/or their segments. Specifically, clarify the intended use of the "typ" Header Parameter and the "typ" claim, whether they convey syntax or semantics, and indeed, whether this is the right approach. Also clarify the relationship between these type values and MIME (Freed, N. and N. Borenstein, "Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies," November 1996.) [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?

* The "x5t" parameter is currently specified as "a base64url encoded SHA-256 thumbprint of the DER encoding of an X.509 certificate". SHA-1 was traditionally used for certificate digests but collisions are possible to create and can be used for denial of service attacks within multi-tenant services. We need to understand the compatibility issues of using SHA-256 thumbprints instead. We also likely want to specify the digest algorithm explicitly.

* 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. I believe therefore, that we should consider changing the MUST for RSA SHA-256 to RECOMMENDED.

*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.

*Generalize the normative text on signing algorithms so that the descriptions apply equally to the use of various key lengths - not just HMAC SHA-256, RSA SHA-256, and ECDSA P-256 SHA-256.

*Add a table cross-referencing the algorithm name strings used in standard software packages and specifications.

*Add Security Considerations text on timing attacks.

*Finish the Security Considerations section.

*Sort out what to do with the IANA registries if this is first standardized as an OpenID specification.

*Write the related specification for encoding public keys using JSON, as per the agreement documented at http://self-issued.info/?p=390. This will be used by the "jku" (JSON Key URL) header parameter.

*Write the companion encryption specification, per the agreements documented at http://self-issued.info/?p=378.

---

13. References

TOC

13.1. 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.
[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).
[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).
[RFC5280]	Cooper, D., Santesson, S., Farrell, S., Boeyen, S., Housley, R., and W. Polk, “Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile,” RFC 5280, May 2008 (TXT).
[USA15]	Davis, M., Whistler, K., and M. Dürst, “Unicode Normalization Forms,” Unicode Standard Annex 15, 09 2009.

13.2. 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.

Appendix A. JWT Examples

TOC

A.1. JWT using HMAC SHA-256

TOC

A.1.1. Encoding

TOC

The Decoded JWT Payload 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 Claims Objects and no canonicalization is performed before encoding. The

following byte array contains the UTF-8 characters for the Decoded JWT Payload 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 Payload Segment value:

eyJpc3MiOiJqb2UiLA0KICJleHAiOjEzMDA4MTkzODAsDQogImh0dHA6Ly9leGFtcGxlLmNvbS9pc19yb290IjpC

The following example JSON header object declares that the data structure is a JSON Web Token (JWT) and the JWT Signing Input 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 Header 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 Header Segment value:

eyJ0eXAiOiJKV1QiLA0KICJhbGciOiJIUzI1NiJ9

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

eyJ0eXAiOiJKV1QiLA0KICJhbGciOiJIUzI1NiJ9
.
eyJpc3MiOiJqb2UiLA0KICJleHAiOjEzMDA4MTkzODAsDQogImh0dHA6Ly9leGFtcGxlLmNvbS9pc19yb290IjpC

The UTF-8 representation of the JWT 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 used 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
JWT 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 JWT Crypto Segment
value:

```
dBjftJeZ4CVP-mB92K27uhbUJU1p1r_wW1gFWFOEjXk
```

Combining these segments in the order Header.Payload.Signature with
period characters between the segments yields this complete JWT using
the JWT Compact Serialization (with line breaks for display purposes
only):

```
eyJ0eXAiOiJKV1QiLA0KICJhbGciOiJIUzI1NiJ9
.
eyJpc3MiOiJqb2UiLA0KICJleHAiOjEzMDA4MTkzODAsDQogImh0dHA6Ly9leGFtcGxlLmNvbS9pc19yb290IjpC
.
dBjftJeZ4CVP-mB92K27uhbUJU1p1r_wW1gFWFOEjXk
```

A.1.2. Decoding

TOC

Decoding the JWT first requires removing the base64url encoding from
the JWT Header Segment, the JWT Payload Segment, and the JWT Crypto
Segment. We base64url decode the segments per Section 7 (Base64url
encoding as used by JWTs) and turn them into the corresponding byte
arrays. We translate the header segment byte array containing UTF-8
encoded characters into the Decoded JWT Header Segment string.
Likewise, if the payload represents a JWT Claims Object, we translate
the payload segment byte array containing UTF-8 encoded characters into
a Decoded JWT Claims Object 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 JWT Crypto Segment. If any of the validation steps fail, the token
MUST be rejected.

First, we validate that the decoded JWT Header Segment string is legal JSON.
If the payload represents a JWT Claims Object, we also validate that the decoded JWT Payload Segment 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 JWT Signing Input as input to a SHA-256 HMAC function and then taking the output and determining if it matches the Decoded JWT Crypto Segment. If it matches exactly, the token has been validated.

A.2. JWT using RSA SHA-256

TOC

A.2.1. Encoding

TOC

The Decoded JWT Payload 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 Payload Segment will therefore be the same, its computation is not repeated here. However, the Decoded JWT Header 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 Header Segment used is:

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

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

```
eyJhbGciOiJSUzI1NiJ9
```

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

```
eyJhbGciOiJSUzI1NiJ9
.
eyJpc3MiOiJqb2UiLA0KICJleHAiOjEzMDA4MTkzODAsDQogImh0dHA6Ly9leGFtcGxlLmNvbS9pc19yb290IjpU
```

The UTF-8 representation of the JWT 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 JWT 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 JWT Crypto Segment:

cC4hiUPoj9Eetdgtv3hF80EGrhuB__dzERat0XF9g2VtQgr9PJbu3XOiZj5RZmh7AAuHIm4Bh-0Qc_lF5YKt_O8W

Combining these segments in the order Header.Payload.Signature with period characters between the segments yields this complete JWT using the JWT Compact Serialization (with line breaks for display purposes only):

```
eyJhbGciOiJSUzI1NiJ9
.
eyJpc3MiOiJqb2UiLA0KICJleHAiOjEzMDA4MTkzODAsDQogImh0dHA6Ly9leGFtcGxlLmNvbS9pc19yb290IjpU
.
cC4hiUPoj9Eetdgtv3hF80EGrhuB__dzERat0XF9g2VtQgr9PJbu3XOiZj5RZmh7AAuHIm4Bh-0Qc_lF5YKt_O8W
```

A.2.2. Decoding

TOC

Decoding the JWT from this example requires processing the JWT Header Segment and JWT Payload Segment 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 JWT Crypto Segment. If any of the validation steps fail, the token MUST be rejected.
First, we validate that the decoded JWT Header Segment string is legal JSON.
If the payload represents a JWT Claims Object, we also validate that the decoded JWT Payload Segment string is 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 UTF-8 representation of the JWT Signing Input to an RSA signature verifier that has been configured to use the SHA-256 hash function.

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

TOC

A.3.1. Encoding

TOC

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

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

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

```
eyJhbGciOiJFUzI1NiJ9
```

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

```
eyJhbGciOiJFUzI1NiJ9
.
eyJpc3MiOiJqb2UiLA0KICJleHAiOjEzMDA4MTkzODAsDQogImh0dHA6Ly9leGFtcGxlLmNvbS9pc19yb290IjpO
```

The UTF-8 representation of the JWT 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 JWT 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 JWT Crypto Segment:

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

Combining these segments in the order Header.Payload.Signature with period characters between the segments yields this complete JWT using the JWT Compact Serialization (with line breaks for display purposes only):

```
eyJhbGciOiJFUzI1NiJ9
.
eyJpc3MiOiJqb2UiLA0KICJleHAiOjEzMDA4MTkzODAsDQogImh0dHA6Ly9leGFtcGxlLmNvbS9pc19yb290IjpC
.
DtEhU3ljbEg8L38VWAfUAqOyKAM6-Xx-F4GawxaepmXFCgfTjDxw5djxLa8ISlSApmWQxfKTUJqPP3-Kg6NU1Q
```

A.3.2. Decoding

TOC

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

A.3.3. Validating

TOC

Since the "alg" parameter in the header is "ES256", we validate the ECDSA P-256 SHA-256 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 JWT Header Segment string is legal JSON.
If the payload represents a JWT Claims Object, we also validate that the decoded JWT Payload Segment string is 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 UTF-8 representation of the JWT 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 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.

A.4. JWT using JSON Serialization

TOC

Previous example JWTs shown have used the JWT Compact Serialization. This section contains an example JWT using the JWT JSON Serialization. This example demonstrates the capability for conveying multiple signatures for the same JWT.

A.4.1. Encoding

TOC

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

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

Two signatures are used in this JWT: an RSA SHA-256 signature, for which the header and signature values are the same as in Appendix A.2 (JWT using RSA SHA-256), and an ECDSA P-256 SHA-256 signature, for which the header and signature values are the same as in Appendix A.3

(JWT using ECDSA P-256 SHA-256). The two Decoded JWT Header Segments used are:

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

and:

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

Since the computations for all JWT Token Segments used in this example were already presented in previous examples, they are not repeated here.
A JSON Serialization of this JWT is as follows:

```
{"header":[
  "eyJhbGciOiJSUzI1NiJ9",
  "eyJhbGciOiJFUzI1NiJ9"],
 "payload":"eyJpc3MiOiJqb2UiLA0KICJleHAiOjEzMDA4MTkzODAsDQogImh0dHA6Ly9leGFtcGxlLmNvbS9p
 "signature":[
  "cC4hiUPoj9Eetdgtv3hF80EGrhuB__dzERat0XF9g2VtQgr9PJbu3XOiZj5RZmh7AAuHIm4Bh-0Qc_lF5YKt_
  "DtEhU3ljbEg8L38VWAfUAqOyKAM6-Xx-F4GawxaepmXFCgfTjDxw5djxLa8ISlSApmWQxfKTUJqPP3-Kg6NU1
}
```

A.4.2. Decoding

TOC

Decoding the JWT first requires removing the base64url encoding from the array of JWT Header Segments, the JWT Payload Segment, and the array of JWT Crypto Segments. We base64url decode the segments per Section 7 (Base64url encoding as used by JWTs) and turn them into the corresponding byte arrays. We translate the header segment byte arrays containing UTF-8 encoded characters into Decoded JWT Header Segment strings. Likewise, if the payload represents a JWT Claims Object, we translate the payload segment byte array into a Decoded JWT Claims Object string.

A.4.3. Validating

TOC

If any of the validation steps fail, the token MUST be rejected.
First, we validate that the header and signature arrays contain the same number of elements.
Next, we validate that the Decoded JWT Header Segment strings are all legal JSON.

If the payload represents a JWT Claims Object, we also validate that the decoded JWT Payload Segment string is legal JSON.
Finally, for each Decoded JWT Header Segment, we validate the corresponding signature using the algorithm specified in the "alg" parameter, which must be present.

Appendix B. Notes on implementing base64url encoding without padding

TOC

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

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

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

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

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

```
3 236 255 224 193


A-z_4ME
```

Appendix C. 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.

Appendix D. Relationship of JWTs to Simple Web Tokens (SWTs)

TOC

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. The signed content of a SWT must be a set of claims, whereas the payload of a JWT, in general, can be any base64url encoded content.

Appendix E. 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.

Appendix F. Document History

TOC

-01

*Draft incorporating consensus decisions reached at IIW.

-00

*Public draft published before November 2010 IIW based upon the JSON token convergence proposal incorporating input from several implementers of related specifications.

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
	Paul Tarjan
	Facebook
Email:	paul.tarjan@facebook.com