Network Working Group	M.B. Jones
Internet-Draft	Microsoft
Intended status: Standards Track	D. Balfanz
Expires: May 03, 2012	Google
	J. Bradley
	independent
	Y.Y. Goland
	Microsoft
	J. Panzer
	Google
	N. Sakimura
	Nomura Research Institute
	P. Tarjan
	Facebook
	October 31, 2011

JSON Web Token (JWT)
draft-jones-json-web-token-06

Abstract

JSON Web Token (JWT) is a means of representing claims to be transferred between two parties. The claims in a JWT are encoded as a JSON object that is digitally signed using JSON Web Signature (JWS) and/or encrypted using JSON Web Encryption (JWE).
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 *[RFC2119]*.

Status of this Memo

Copyright Notice

Table of Contents

1. Introduction

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 *[RFC4627]*) that is base64url encoded and digitally signed and/or encrypted. Signing is accomplished using JSON Web Signature (JWS) [JWS]. Encryption is accomplished using JSON Web Encryption (JWE) [JWE].
The suggested pronunciation of JWT is the same as the English word "jot".

2. Terminology

JSON Web Token (JWT) A string consisting of three parts: the JWT Header, the JWT Second Part, and the JWT Third Part, in that order, with the parts being separated by period ('.') characters, and each part containing base64url encoded content.

JWT Header A string containing a JSON object that describes the cryptographic operations applied to the JWT. When the JWT is signed, the JWT Header is the JWS Header. When the JWT is encrypted, the JWT Header is the JWE Header.

JWT Second Part When the JWT is signed, the JWT Second Part is the Encoded JWS Payload. When the JWT is encrypted, the JWT Second Part is the Encoded JWE Encrypted Key.

JWT Third Part When the JWT is signed, the JWT Third Part is the Encoded JWS Signature. When the JWT is encrypted, the JWT Third Part is the Encoded JWE Ciphertext.

JWT Claims Object A JSON object that represents the claims contained in the JWT. When the JWT is signed, the bytes of the UTF-8 representation of the JWT Claims Object are base64url encoded to create the Encoded JWS Payload. When the JWT is encrypted, the bytes

of the UTF-8 representation of the JWT Claims Object are used as the JWE Plaintext.

Claim Names The names of the members of the JWT Claims Object.

Claim Values The values of the members of the JWT Claims Object.

Encoded JWT Header Base64url encoding of the bytes of the UTF-8 RFC 3629 *[RFC3629]* representation of the JWT Header.

Base64url Encoding For the purposes of this specification, this term always refers to the URL- and filename-safe Base64 encoding described in RFC 4648 *[RFC4648]*, Section 5, with the (non URL-safe) '=' padding characters omitted, as permitted by Section 3.2. (See Appendix C of [JWS] for notes on implementing base64url encoding without padding.)

3. JSON Web Token (JWT) Overview

JWTs represent a set of claims as a JSON object that is base64url encoded and digitally signed and/or encrypted. This object is the JWT Claims Object. As per RFC 4627 *[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 member names within the JWT Claims Object are referred to as Claim Names. These names MUST be unique. The corresponding values are referred to as Claim Values.
The bytes of the UTF-8 representation of the JWT Claims Object are signed in the manner described in JSON Web Signature (JWS) [JWS] and/or encrypted in the manner described in JSON Web Encryption (JWE) [JWE]. The contents of the JWT Header describe the cryptographic operations applied to the JWT Claims Object. If the JWT Header is a JWS Header, the claims are signed. If the JWT Header is a JWE Header, the claims are encrypted.
A JWT is represented as the concatenation of the Encoded JWT Header, the JWT Second Part, and the JWT Third Part, in that order, with the parts being separated by period ('.') characters. When signed, the three parts of the JWT are the three parts of a JWS used to represent the JWT. When encrypted, the three parts of the JWT are the three parts of a JWE used to represent the JWT.

3.1. Example JWT

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

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

Base64url encoding the bytes of the UTF-8 representation of the JWT Header yields this Encoded JWS Header value, which is used as the Encoded JWT Header:

```
eyJ0eXAiOiJKV1QiLA0KICJhbGciOiJIUzI1NiJ9
```

The following is an example of a JWT Claims Object:

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

Base64url encoding the bytes of the UTF-8 representation of the JSON Claims Object yields this Encoded JWS Payload, which is used as the JWT Second Part:

```
eyJpc3MiOiJqb2UiLA0KICJleHAiOjEzMDA4MTkzODAsDQogImh0dHA6Ly9leGFtcGxlLmNvbS9pc19yb290Ijp0cnVl
```

Signing the Encoded JWS Header and Encoded JWS Payload with the HMAC SHA-256 algorithm and base64url encoding the signature in the manner specified in [JWS], yields this Encoded JWS Signature, which is used as the JWT Third Part:

```
dBjftJeZ4CVP-mB92K27uhbUJU1p1r_wW1gFWFOEjXk
```

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

```
eyJ0eXAiOiJKV1QiLA0KICJhbGciOiJIUzI1NiJ9
.
eyJpc3MiOiJqb2UiLA0KICJleHAiOjEzMDA4MTkzODAsDQogImh0dHA6Ly9leGFtcGxlLmNvbS9pc19yb290Ijp0cnVl
.
dBjftJeZ4CVP-mB92K27uhbUJU1p1r_wW1gFWFOEjXk
```

This computation is illustrated in more detail in [JWS], Appendix A.1.

4. JWT Claims

A JWT contains a set of claims represented as a base64url encoded JSON object. 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

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 to be compact.

Claim Name	JSON Value Type	Claim Syntax	Claim Semantics
exp	number	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.
nbf	number	IntDate	The nbf (not before) claim identifies the time before which the token MUST NOT be accepted for processing. The processing of the nbf claim requires that the current date/time MUST be after or equal to the not-before date/time listed in the nbf claim. Implementers MAY provide for some small leeway, usually no more than a few minutes, to account for clock skew. This claim is OPTIONAL.
iat	number	IntDate	The iat (issued at) claim identifies the time at which the JWT was issued. This claim can be used to determine the age of the token. This claim is OPTIONAL.
iss	string	StringOrURI	The iss (issuer) claim identifies the principal that issued the JWT. The processing of this claim is generally application specific. The iss value is case sensitive. This claim is OPTIONAL.
aud	string	StringOrURI	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

Claim Name	JSON Value Type	Claim Syntax	Claim Semantics
			of the audience value is generally application specific. The aud value is case sensitive. This claim is OPTIONAL.
typ	string	String	The typ (type) claim is used to declare a type for the contents of this JWT Claims Object. The typ value is case sensitive. This claim is OPTIONAL.

Reserved Claim Definitions

Additional reserved claim names MAY be defined via the IANA JSON Web Token Claims registry, as per Section 9. The syntax values used above are defined as follows:

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

Claim Syntax Definitions

4.2. Public Claim Names

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 *[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

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

5. JWT Header

The members of the JSON object represented by the JWT Header describe the cryptographic operations applied to the JWT and optionally, additional properties of the JWT.
There are two ways of distinguishing whether the JWT is a JWS or JWE. The first is by examining the alg (algorithm) header value. If the value represents a signature algorithm, the JWT is a JWS; if it represents an encryption algorithm, the JWT is a JWE. A second method is determining whether an enc (encryption method) member exists. If the enc member exists, the JWT is a JWE; otherwise, the JWT is a JWS. Both methods will yield the same result.
Implementations MUST understand the entire contents of the header; otherwise, the JWT MUST be rejected for processing.
JWS Header Parameters are defined by [JWS]. JWE Header Parameters are defined by [JWE]. This specification further specifies the use of the following header parameters in both the cases where the JWT is a JWS and where it is a JWE.

Header Parameter Name	JSON Value Type	Header Parameter Syntax	Header Parameter Semantics
typ	string	String	The typ (type) header parameter is used to declare structural information about the JWT. In the normal case where nested signing or encryption operations are not employed, the use of this header parameter is OPTIONAL, and if present, it is RECOMMENDED that its value be either "JWT" or "http://openid.net/specs/jwt/1.0". In the case that nested signing or encryption steps are employed, the use of this header parameter is REQUIRED; in this case, the value MUST either be "JWS", to indicate that a nested signed JWT is carried in this JWT or "JWE", to indicate that a nested encrypted JWT is carried in this JWT.

Reserved Header Parameter Usage

6. Plaintext JWTs

To support use cases where the JWT content is secured by a means other than a signature and/or encryption contained within the token (such as a signature on a data structure containing the token), JWTs MAY also be created without a signature or encryption. Plaintext JWTs MUST use the alg value none, and are formatted identically to a signed JWT with an empty signature. This means that the base64url encoding of the bytes representing the UTF-8 encoding of the JWT Claims Object is the JWT Second Part, and the empty string is the JWT Third Part.

6.1. Example Plaintext JWT

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

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

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

```
eyJhbGciOiJub25lIn0
```

The following is an example of a JWT Claims Object:

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

Base64url encoding the bytes of the UTF-8 representation of the JSON Claims Object yields this Encoded JWS Payload, which is used as the JWT Second Part:

```
eyJpc3MiOiJqb2UiLA0KICJleHAiOjEzMDA4MTkzODAsDQogImh0dHA6Ly9leGFtcGxlLmNvbS9pc19yb290Ijp0cnVl
```

The JWT Third Part is the empty string.
Concatenating these parts in the order Header.Second.Third with period characters between the parts yields this complete JWT (with line breaks for display purposes only):

```
eyJhbGciOiJub25lIn0
.
eyJpc3MiOiJqb2UiLA0KICJleHAiOjEzMDA4MTkzODAsDQogImh0dHA6Ly9leGFtcGxlLmNvbS9pc19yb290Ijp0cnVl
.
```

7. Rules for Creating and Validating a JWT

To create a JWT, one MUST follow these steps:

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

2. Let the Message be the bytes of the UTF-8 representation of the JWT Claims Object.

3. Create a JWT Header containing the desired set of header parameters. If the JWT is to be signed or encrypted, they MUST conform to either the [JWS] or [JWE] specifications, respectively. Else, if the JWT is to be plaintext, the alg value none MUST be used. Note that white space is explicitly allowed in the representation and no canonicalization is performed before encoding.

4. Base64url encode the bytes of the UTF-8 representation of the JWT Header. Let this be the Encoded JWT Header.

5. Depending upon whether the JWT is to be signed, encrypted, or plaintext, there are three cases:

   * If the JWT is to be signed, create a JWS using the JWT Header as the JWS Header and the Message as the JWS Payload; all steps specified in [JWS] for creating a JWS MUST be followed. Let the JWT Second Part be the Encoded JWS Payload and let the JWT Third Part be the Encoded JWS Signature.

   * If the JWT is to be encrypted, create a JWE using the JWT Header as the JWE Header and the Message as the JWE Plaintext; all steps specified in [JWE] for creating a JWE MUST be followed. Let the JWT Second Part be the Encoded JWE Encrypted Key and let the JWT Third Part be the Encoded JWS Ciphertext.

   * Else, if the JWT is to be plaintext, let the JWT Second Part be the base64url encoding of the Message and let the JWT Third Part be the empty string.

6. Concatenate the Encoded JWT Header, the JWT Second Part, and the JWT Third Part in that order, separating each by period ('.') characters.

7. If a nested signing or encryption operation will be performed, let the Message be this concatenation, and return to Step 3, using a typ value of either "JWS" or "JWE" respectively in the new JWT Header created in that step.

8. Otherwise, let the resulting JWT be this concatenation.

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 exactly two period characters.

2. The JWT MUST be split on the two period characters resulting in three strings. The first string is the Encoded JWT Header; the second is the JWT Second Part; the third is the JWT Third Part.

3. The Encoded JWT Header MUST be successfully base64url decoded following the restriction given in this specification that no padding characters have been used.

4. The JWT Header MUST be completely valid JSON syntax conforming to RFC 4627 *[RFC4627]*.

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

6. Determine whether the JWT is signed, encrypted, or plaintext by examining the alg (algorithm) header value and optionally, the enc (encryption method) header value, if present.

7. Depending upon whether the JWT signed, encrypted, or plaintext, there are three cases:

   * If the JWT is signed, all steps specified in [JWS] for validating a JWS MUST be followed. Let the Message be the result of base64url decoding the JWS Payload.

   * If the JWT is encrypted, all steps specified in [JWE] for validating a JWE MUST be followed. Let the Message be the JWE Plaintext.

   * Else, if the JWT is plaintext, let the Message be the result of base64url decoding the JWE Second Part. The Third Part MUST be verified to be the empty string.

8. If the JWT Header contains a typ value of either "JWS" or "JWE", then the Message contains a JWT that was the subject of nested signing or encryption operations, respectively. In this case, return to Step 1, using the Message as the JWT.

9. Otherwise, let the JWT Claims object be the Message.

10. The JWT Claims Object MUST be completely valid JSON syntax conforming to RFC 4627 *[RFC4627]*.

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

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 JWT Header to see if there is a matching header parameter name. A similar process occurs when determining if the value of the alg header parameter represents a supported algorithm. Comparing Unicode strings, however, has significant security implications, as per Section 10. Comparisons between JSON strings and other Unicode strings MUST be performed as specified below:

1. Remove any JSON applied escaping to produce an array of Unicode code points.

2. Unicode Normalization *[USA15]* MUST NOT be applied at any point to either the JSON string or to the string it is to be compared against.

3. Comparisons between the two strings MUST be performed as a Unicode code point to code point equality comparison.

8. Cryptographic Algorithms

JWTs use JSON Web Signature (JWS) [JWS] and JSON Web Encryption (JWE) [JWE] to sign and/or encrypt the contents of the JWT.
Of the JWS signing algorithms, only HMAC SHA-256 MUST be implemented by conforming JWT implementations. It is RECOMMENDED that implementations also support the RSA SHA-256 and ECDSA P-256 SHA-256 algorithms. Support for other algorithms and key sizes is OPTIONAL.
If an implementation provides encryption capabilities, of the JWE encryption algorithms, only RSA-PKCS1-1.5 with 2048 bit keys, AES-128-CBC, and AES-256-CBC MUST be implemented by conforming implementations. It is RECOMMENDED that implementations also support ECDH-ES with 256 bit keys, AES-128-GCM, and AES-256-GCM. Support for other algorithms and key sizes is OPTIONAL.

9. IANA Considerations

This specification calls for:

*A new IANA registry entitled "JSON Web Token Claims" for reserved claim names is defined in Section 4.1. Inclusion in the registry is RFC Required in the RFC 5226 *[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.

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.
TBD: Need to put in text about why strict JSON validation is necessary. Basically, that if malformed JSON is received then the intent of the sender is impossible to reliably discern. One example of malformed JSON that MUST be rejected is an object in which the same member name occurs multiple times. 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.
TBD: Write about the need to secure the token content if a signature is not contained in the JWT itself.

10.1. Unicode Comparison Security Issues

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 *[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.

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

The following items remain to be done in this draft:

- *Provide an example of an encrypted JWT.

*Clarify the optional ability to provide type information for JWTs and/or their parts. Specifically, clarify whether we need to specify the typ Claim Name in addition to the Header Parameter, whether it conveys syntax or semantics, and indeed, whether this is the right approach. Also clarify the relationship between these type values and MIME *[RFC2045]* types.

*Think about how to best describe the concept currently described as "the bytes of the UTF-8 representation of". Possible terms to use instead of "bytes of" include "byte sequence", "octet series", and "octet sequence". Also consider whether we want to add an overall clarifying statement somewhere in each spec something like "every place we say 'the UTF-8 representation of X', we mean 'the bytes of the UTF-8 representation of X'". That would potentially allow us to omit the "the bytes of" part everywhere else.

*Consider whether a media type should be proposed, such as "application/jwt".

*Finish the Security Considerations section.

*Possibly write a companion specification that contains the former JWT JSON Serialization.

12. References

12.1. Normative References

[RFC2045]	Freed, N. and N.S. Borenstein, "Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies", RFC 2045, November 1996.
[RFC2119]	Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.
[RFC3339]	Klyne, G. and C. Newman, "Date and Time on the Internet: Timestamps", RFC 3339, July 2002.
[RFC3629]	Yergeau, F., "UTF-8, a transformation format of ISO 10646", STD 63, RFC 3629, November 2003.
[RFC3986]	Berners-Lee, T., Fielding, R. and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax", STD 66, RFC 3986, January 2005.
[RFC4627]	Crockford, D., "The application/json Media Type for JavaScript Object Notation (JSON)", RFC 4627, July 2006.
[RFC4648]	Josefsson, S., "The Base16, Base32, and Base64 Data Encodings", RFC 4648, October 2006.
[RFC5226]	Narten, T. and H. Alvestrand, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 5226, May 2008.

[USA15]	Davis, M., Whistler, K. and M. Dürst, "Unicode Normalization Forms", Unicode Standard Annex 15, 09 2009.
[JWS]	Jones, M.B., Balfanz, D., Bradley, J., Goland, Y.Y., Panzer, J., Sakimura, N. and P. Tarjan, "JSON Web Signature (JWS)", October 2011.

12.2. Informative References

[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.
[W3C.CR-xml11-20021015]	Cowan, J, "Extensible Markup Language (XML) 1.1", W3C CR CR-xml11-20021015, October 2002.
[RFC3275]	Eastlake, D., Reagle, J. and D. Solo, "(Extensible Markup Language) XML-Signature Syntax and Processing", RFC 3275, March 2002.
[RFC4122]	Leach, P., Mealling, M. and R. Salz, "A Universally Unique IDentifier (UUID) URN Namespace", RFC 4122, July 2005.
[SWT]	Hardt, D. and Y.Y. Goland, "Simple Web Token (SWT)", Version 0.9.5.1, November 2009.
[MagicSignatures]	Panzer (editor), J., Laurie, B. and D. Balfanz, "Magic Signatures", August 2010.
[JSS]	Bradley, J. and N. Sakimura (editor), "JSON Simple Sign", September 2010.
[CanvasApp]	Facebook, , "Canvas Applications", 2010.
[JWE]	Jones, M.B., Rescorla, E. and J. Hildebrand, "JSON Web Encryption (JWE)", October 2011.

Appendix A. Relationship of JWTs to SAML Tokens

SAML 2.0 *[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 *[W3C.CR-xml11-20021015]* and XML DSIG *[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 *[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 B. Relationship of JWTs to Simple Web Tokens (SWTs)

Both JWTs and Simple Web Tokens SWT *[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.

Appendix C. Acknowledgements

The authors acknowledge that the design of JWTs was intentionally influenced by the design and simplicity of Simple Web Tokens *[SWT]* and ideas for JSON tokens that Dick Hardt discussed within the OpenID community.
Solutions for signing JSON content were previously explored by Magic Signatures *[MagicSignatures]*, JSON Simple Sign *[JSS]*, and Canvas Applications *[CanvasApp]*, all of which influenced this draft.

Appendix D. Document History

-06

*Reference and use content from [JWS] and [JWE], rather than repeating it here.

*Simplified terminology to better match JWE, where the terms "JWT Header" and "Encoded JWT Header" are now used, for instance, rather than the previous terms "Decoded JWT Header Segment" and "JWT Header Segment". Also changed to "Plaintext JWT" from "Unsigned JWT".

*Describe how to perform nested encryption and signing operations.

*Changed "integer" to "number", since that is the correct JSON type.

*Changed StringAndURI to StringOrURI.

-05

*Added the nbf (not before) claim and clarified the meaning of the iat (issued at) claim.

-04

*Correct typo found by John Bradley: "the JWT Claim Segment is the empty string" -> "the JWT Crypto Segment is the empty string".

-03

*Added "http://openid.net/specs/jwt/1.0" as a token type identifier URI for JWTs.

*Added iat (issued at) claim.

*Changed RSA SHA-256 from MUST be supported to RECOMMENDED that it be supported. Rationale: Several people have objected to the requirement for implementing RSA SHA-256, some because they will only be using HMACs and symmetric keys, and others because they only want to use ECDSA when using asymmetric keys, either for security or key length reasons, or both.

*Defined alg value none to represent unsigned JWTs.

-02

*Split signature specification out into separate draft-jones-json-web-signature-00. This split introduced no semantic changes.

*The JWT Compact Serialization is now the only token serialization format specified in this draft. The JWT JSON Serialization can continue to be defined in a companion specification.

-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

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

Dirk Balfanz Balfanz Google EMail: balfanz@google.com

John Bradley Bradley independent EMail: ve7jtb@ve7jtb.com

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

John Panzer Panzer Google EMail: jpanzer@google.com

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

Paul Tarjan Tarjan Facebook EMail: pt@fb.com