Internet Engineering Task Force (IETF)
Request for Comments: 7519
Category: Standards Track
ISSN: 2070-1721

M. Jones
Microsoft
J. Bradley
Ping Identity
N. Sakimura
NRI

May 2015

# JSON Web Token (JWT)

## Abstract

JSON Web Token (JWT) is a compact, URL-safe means of representing claims to be transferred between two parties. The claims in a JWT are encoded as a JSON object that is used as the payload of a JSON Web Signature (JWS) structure or as the plaintext of a JSON Web Encryption (JWE) structure, enabling the claims to be digitally signed or integrity protected with a Message Authentication Code (MAC) and/or encrypted.

## Status of This Memo

## Copyright Notice

## Table of Contents

## 1. Introduction

JSON Web Token (JWT) is a compact claims representation format intended for space constrained environments such as HTTP Authorization headers and URI query parameters. JWTs encode claims to be transmitted as a JSON [RFC7159] object that is used as the payload of a JSON Web Signature (JWS) [JWS] structure or as the plaintext of a JSON Web Encryption (JWE) [JWE] structure, enabling the claims to be digitally signed or integrity protected with a Message Authentication Code (MAC) and/or encrypted. JWTs are always represented using the JWS Compact Serialization or the JWE Compact Serialization.

The suggested pronunciation of JWT is the same as the English word "jot".

### 1.1. Notational Conventions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in "Key words for use in RFCs to Indicate Requirement Levels" [RFC2119]. The interpretation should only be applied when the terms appear in all capital letters.

## 2. Terminology

The terms "JSON Web Signature (JWS)", "Base64url Encoding", "Header Parameter", "JOSE Header", "JWS Compact Serialization", "JWS Payload", "JWS Signature", and "Unsecured JWS" are defined by the JWS specification [JWS].

The terms "JSON Web Encryption (JWE)", "Content Encryption Key (CEK)", "JWE Compact Serialization", "JWE Encrypted Key", and "JWE Initialization Vector" are defined by the JWE specification [JWE].

The terms "Ciphertext", "Digital Signature", "Message Authentication Code (MAC)", and "Plaintext" are defined by the "Internet Security Glossary, Version 2" [RFC4949].

These terms are defined by this specification:

JSON Web Token (JWT)

> A string representing a set of claims as a JSON object that is encoded in a JWS or JWE, enabling the claims to be digitally signed or MACed and/or encrypted.

JWT Claims Set

> A JSON object that contains the claims conveyed by the JWT.

Claim

> A piece of information asserted about a subject. A claim is represented as a name/value pair consisting of a Claim Name and a Claim Value.

Claim Name

> The name portion of a claim representation. A Claim Name is always a string.

Claim Value

> The value portion of a claim representation. A Claim Value can be any JSON value.

Nested JWT

> A JWT in which nested signing and/or encryption are employed. In Nested JWTs, a JWT is used as the payload or plaintext value of an enclosing JWS or JWE structure, respectively.

Unsecured JWT

A JWT whose claims are not integrity protected or encrypted.

Collision-Resistant Name

A name in a namespace that enables names to be allocated in a manner such that they are highly unlikely to collide with other names. Examples of collision-resistant namespaces include: Domain Names, Object Identifiers (OIDs) as defined in the ITU-T X.660 and X.670 Recommendation series, and Universally Unique IDentifiers (UUIDs) [RFC4122]. When using an administratively delegated namespace, the definer of a name needs to take reasonable precautions to ensure they are in control of the portion of the namespace they use to define the name.

StringOrURI

A JSON string value, with the additional requirement that while arbitrary string values MAY be used, any value containing a ":" character MUST be a URI [RFC3986]. StringOrURI values are compared as case-sensitive strings with no transformations or canonicalizations applied.

NumericDate

A JSON numeric value representing the number of seconds from 1970-01-01T00:00:00Z UTC until the specified UTC date/time, ignoring leap seconds. This is equivalent to the IEEE Std 1003.1, 2013 Edition [POSIX.1] definition "Seconds Since the Epoch", in which each day is accounted for by exactly 86400 seconds, other than that non-integer values can be represented. See RFC 3339 [RFC3339] for details regarding date/times in general and UTC in particular.

## 3. JSON Web Token (JWT) Overview

JWTs represent a set of claims as a JSON object that is encoded in a JWS and/or JWE structure. This JSON object is the JWT Claims Set. As per Section 4 of RFC 7159 [RFC7159], 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. This JSON object MAY contain whitespace and/or line breaks before or after any JSON values or structural characters, in accordance with Section 2 of RFC 7159 [RFC7159].

The member names within the JWT Claims Set are referred to as Claim Names. The corresponding values are referred to as Claim Values.

The contents of the JOSE Header describe the cryptographic operations applied to the JWT Claims Set. If the JOSE Header is for a JWS, the JWT is represented as a JWS and the claims are digitally signed or MACed, with the JWT Claims Set being the JWS Payload. If the JOSE Header is for a JWE, the JWT is represented as a JWE and the claims are encrypted, with the JWT Claims Set being the plaintext encrypted by the JWE. A JWT may be enclosed in another JWE or JWS structure to create a Nested JWT, enabling nested signing and encryption to be performed.

A JWT is represented as a sequence of URL-safe parts separated by period ('.') characters. Each part contains a base64url-encoded value. The number of parts in the JWT is dependent upon the representation of the resulting JWS using the JWS Compact Serialization or JWE using the JWE Compact Serialization.

## 3.1. Example JWT

The following example JOSE Header declares that the encoded object is a JWT, and the JWT is a JWS that is MACed using the HMAC SHA-256 algorithm:

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

To remove potential ambiguities in the representation of the JSON object above, the octet sequence for the actual UTF-8 representation used in this example for the JOSE Header above is also included below. (Note that ambiguities can arise due to differing platform representations of line breaks (CRLF versus LF), differing spacing at the beginning and ends of lines, whether the last line has a terminating line break or not, and other causes. In the representation used in this example, the first line has no leading or trailing spaces, a CRLF line break (13, 10) occurs between the first and second lines, the second line has one leading space (32) and no trailing spaces, and the last line does not have a terminating line break.) The octets representing the UTF-8 representation of the JOSE Header in this example (using JSON array notation) are:

[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 the octets of the UTF-8 representation of the JOSE Header yields this encoded JOSE Header value:

```
eyJ0eXAiOiJKV1QiLA0KICJhbGciOiJIUzI1NiJ9
```

The following is an example of a JWT Claims Set:

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

The following octet sequence, which is the UTF-8 representation used in this example for the JWT Claims Set above, is the JWS Payload:

[123, 34, 105, 115, 115, 34, 58, 34, 106, 111, 101, 34, 44, 13, 10, 32, 34, 101, 120, 112, 34, 58, 49, 51, 48, 48, 56, 49, 57, 51, 56, 48, 44, 13, 10, 32, 34, 104, 116, 116, 112, 58, 47, 47, 101, 120, 97, 109, 112, 108, 101, 46, 99, 111, 109, 47, 105, 115, 95, 114, 111, 111, 116, 34, 58, 116, 114, 117, 101, 125]

Base64url encoding the JWS Payload yields this encoded JWS Payload (with line breaks for display purposes only):

```
eyJpc3MiOiJqb2UiLA0KICJleHAiOjEzMDA4MTkzODAsDQogImh0dHA6Ly
9leGFtcGxlLmNvbS9pc19yb290Ijp0cnVlfQ
```

Computing the MAC of the encoded JOSE Header and encoded JWS Payload with the HMAC SHA-256 algorithm and base64url encoding the HMAC value in the manner specified in [JWS] yields this encoded JWS Signature:

```
dBjftJeZ4CVP-mB92K27uhbUJU1p1r_wW1gFWFOEjXk
```

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

```
eyJ0eXAiOiJKV1QiLA0KICJhbGciOiJIUzI1NiJ9
.
eyJpc3MiOiJqb2UiLA0KICJleHAiOjEzMDA4MTkzODAsDQogImh0dHA6Ly9leGFt
cGxlLmNvbS9pc19yb290Ijp0cnVlfQ
.
dBjftJeZ4CVP-mB92K27uhbUJU1p1r_wW1gFWFOEjXk
```

This computation is illustrated in more detail in Appendix A.1 of [JWS]. See Appendix A.1 for an example of an encrypted JWT.

## 4. JWT Claims

The JWT Claims Set represents a JSON object whose members are the claims conveyed by the JWT. The Claim Names within a JWT Claims Set MUST be unique; JWT parsers MUST either reject JWTs with duplicate Claim Names or use a JSON parser that returns only the lexically last duplicate member name, as specified in Section 15.12 ("The JSON Object") of ECMAScript 5.1 [ECMAScript].

The set of claims that a JWT must contain to be considered valid is context dependent and is outside the scope of this specification. Specific applications of JWTs will require implementations to understand and process some claims in particular ways. However, in the absence of such requirements, all claims that are not understood by implementations MUST be ignored.

There are three classes of JWT Claim Names: Registered Claim Names, Public Claim Names, and Private Claim Names.

## 4.1. Registered Claim Names

The following Claim Names are registered in the IANA "JSON Web Token Claims" registry established by Section 10.1. None of the claims defined below are intended to be mandatory to use or implement in all cases, but rather they provide a starting point for a set of useful, interoperable claims. Applications using JWTs should define which specific claims they use and when they are required or optional. All the names are short because a core goal of JWTs is for the representation to be compact.

### 4.1.1. "iss" (Issuer) Claim

The `iss` (issuer) claim identifies the principal that issued the JWT. The processing of this claim is generally application specific. The `iss` value is a case-sensitive string containing a StringOrURI value. Use of this claim is OPTIONAL.

### 4.1.2. "sub" (Subject) Claim

The `sub` (subject) claim identifies the principal that is the subject of the JWT. The claims in a JWT are normally statements about the subject. The subject value MUST either be scoped to be locally unique in the context of the issuer or be globally unique. The processing of this claim is generally application specific. The `sub` value is a case-sensitive string containing a StringOrURI value. Use of this claim is OPTIONAL.

### 4.1.3. "aud" (Audience) Claim

The `aud` (audience) claim identifies the recipients that the JWT is intended for. Each principal intended to process the JWT MUST identify itself with a value in the audience claim. If the principal processing the claim does not identify itself with a value in the `aud` claim when this claim is present, then the JWT MUST be rejected. In the general case, the `aud` value is an array of case-sensitive strings, each containing a StringOrURI value. In the special case when the JWT has one audience, the `aud` value MAY be a single case-sensitive string containing a StringOrURI value. The interpretation of audience values is generally application specific. Use of this claim is OPTIONAL.

### 4.1.4. "exp" (Expiration Time) Claim

The `exp` (expiration time) claim identifies the expiration time on or after which the JWT 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. Its value MUST be a number containing a NumericDate value. Use of this claim is OPTIONAL.

### 4.1.5. "nbf" (Not Before) Claim

The `nbf` (not before) claim identifies the time before which the JWT 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. Its value MUST be a number containing a NumericDate value. Use of this claim is OPTIONAL.

### 4.1.6. "iat" (Issued At) Claim

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 JWT. Its value MUST be a number containing a NumericDate value. Use of this claim is OPTIONAL.

### 4.1.7. "jti" (JWT ID) Claim

The `jti` (JWT ID) claim provides a unique identifier for the JWT. The identifier value MUST be assigned in a manner that ensures that there is a negligible probability that the same value will be accidentally assigned to a different data object; if the application uses multiple issuers, collisions MUST be prevented among values produced by different issuers as well. The `jti` claim can be used to prevent the JWT from being replayed. The `jti` value is a case-sensitive string. Use of this claim is OPTIONAL.

## 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 registered in the IANA "JSON Web Token Claims" registry established by Section 10.1 or be a Public Name: a value that contains a Collision-Resistant Name. In each case, the definer of the name or value needs to take reasonable precautions to make sure they are in control of the part of the namespace they use to define the Claim Name.

## 4.3. Private Claim Names

A producer and consumer of a JWT MAY agree to use Claim Names that are Private Names: names that are not Registered Claim Names (Section 4.1) or Public Claim Names (Section 4.2). Unlike Public Claim Names, Private Claim Names are subject to collision and should be used with caution.

## 5. JOSE Header

For a JWT object, the members of the JSON object represented by the JOSE Header describe the cryptographic operations applied to the JWT and optionally, additional properties of the JWT. Depending upon whether the JWT is a JWS or JWE, the corresponding rules for the JOSE Header values apply.

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.

## 5.1. "typ" (Type) Header Parameter

The `typ` (type) Header Parameter defined by [JWS] and [JWE] is used by JWT applications to declare the media type [IANA.MediaTypes] of this complete JWT. This is intended for use by the JWT application when values that are not JWTs could also be present in an application data structure that can contain a JWT object; the application can use this value to disambiguate among the different kinds of objects that might be present. It will typically not be used by applications when it is already known that the object is a JWT. This parameter is ignored by JWT implementations; any processing of this parameter is performed by the JWT application. If present, it is RECOMMENDED that its value be `JWT` to indicate that this object is a JWT. While media type names are not case sensitive, it is RECOMMENDED that `JWT` always be spelled using uppercase characters for compatibility with legacy implementations. Use of this Header Parameter is OPTIONAL.

## 5.2. "cty" (Content Type) Header Parameter

The `cty` (content type) Header Parameter defined by [JWS] and [JWE] is used by this specification to convey structural information about the JWT.

In the normal case in which nested signing or encryption operations are not employed, the use of this Header Parameter is NOT RECOMMENDED. In the case that nested signing or encryption is employed, this Header Parameter MUST be present; in this case, the value MUST be `JWT`, to indicate that a Nested JWT is carried in this JWT. While media type names are not case sensitive, it is RECOMMENDED that `JWT` always be spelled using uppercase characters for compatibility with legacy implementations. See Appendix A.2 for an example of a Nested JWT.

## 5.3. Replicating Claims as Header Parameters

In some applications using encrypted JWTs, it is useful to have an unencrypted representation of some claims. This might be used, for instance, in application processing rules to determine whether and how to process the JWT before it is decrypted.

This specification allows claims present in the JWT Claims Set to be replicated as Header Parameters in a JWT that is a JWE, as needed by the application. If such replicated claims are present, the application receiving them SHOULD verify that their values are identical, unless the application defines other specific processing rules for these claims. It is the responsibility of the application to ensure that only claims that are safe to be transmitted in an unencrypted manner are replicated as Header Parameter values in the JWT.

Section 10.4.1 of this specification registers the `iss` (issuer), `sub` (subject), and `aud` (audience) Header Parameter names for the purpose of providing unencrypted replicas of these claims in encrypted JWTs for applications that need them. Other specifications MAY similarly register other names that are registered Claim Names as Header Parameter names, as needed.

## 6. Unsecured JWTs

To support use cases in which the JWT content is secured by a means other than a signature and/or encryption contained within the JWT (such as a signature on a data structure containing the JWT), JWTs MAY also be created without a signature or encryption. An Unsecured JWT is a JWS using the `alg` Header Parameter value `none` and with the empty string for its JWS Signature value, as defined in the JWA specification [JWA]; it is an Unsecured JWS with the JWT Claims Set as its JWS Payload.

## 6.1. Example Unsecured JWT

The following example JOSE Header declares that the encoded object is an Unsecured JWT:

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

Base64url encoding the octets of the UTF-8 representation of the JOSE Header yields this encoded JOSE Header value:

```
eyJhbGciOiJub25lIn0
```

The following is an example of a JWT Claims Set:

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

Base64url encoding the octets of the UTF-8 representation of the JWT Claims Set yields this encoded JWS Payload (with line breaks for display purposes only):

```
eyJpc3MiOiJqb2UiLA0KICJleHAiOjEzMDA4MTkzODAsDQogImh0dHA6Ly9leGFt
cGxlLmNvbS9pc19yb290Ijp0cnVlfQ
```

The encoded JWS Signature is the empty string.

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

```
eyJhbGciOiJub25lIn0
.
eyJpc3MiOiJqb2UiLA0KICJleHAiOjEzMDA4MTkzODAsDQogImh0dHA6Ly9leGFt
cGxlLmNvbS9pc19yb290Ijp0cnVlfQ
.
```

## 7. Creating and Validating JWTs

## 7.1. Creating a JWT

To create a JWT, the following steps are performed. The order of the steps is not significant in cases where there are no dependencies between the inputs and outputs of the steps.

1. Create a JWT Claims Set containing the desired claims. Note that whitespace is explicitly allowed in the representation and no canonicalization need be performed before encoding.
2. Let the Message be the octets of the UTF-8 representation of the JWT Claims Set.
3. Create a JOSE Header containing the desired set of Header Parameters. The JWT MUST conform to either the [JWS] or [JWE] specification. Note that whitespace is explicitly allowed in the representation and no canonicalization need be performed before encoding.
4. Depending upon whether the JWT is a JWS or JWE, there are two cases:
    - If the JWT is a JWS, create a JWS using the Message as the JWS Payload; all steps specified in [JWS] for creating a JWS MUST be followed.
    - Else, if the JWT is a JWE, create a JWE using the Message as the plaintext for the JWE; all steps specified in [JWE] for creating a JWE MUST be followed.
5. If a nested signing or encryption operation will be performed, let the Message be the JWS or JWE, and return to Step 3, using a `cty` (content type) value of `JWT` in the new JOSE Header created in that step.
6. Otherwise, let the resulting JWT be the JWS or JWE.

## 7.2. Validating a JWT

When validating a JWT, the following steps are performed. The order of the steps is not significant in cases where there are no dependencies between the inputs and outputs of the steps. If any of the listed steps fail, then the JWT MUST be rejected -- that is, treated by the application as an invalid input.

1. Verify that the JWT contains at least one period ('.') character.
2. Let the Encoded JOSE Header be the portion of the JWT before the first period ('.') character.
3. Base64url decode the Encoded JOSE Header following the restriction that no line breaks, whitespace, or other additional characters have been used.
4. Verify that the resulting octet sequence is a UTF-8-encoded representation of a completely valid JSON object conforming to RFC 7159 [RFC7159]; let the JOSE Header be this JSON object.
5. Verify that the resulting JOSE Header includes only parameters and values whose syntax and semantics are both understood and supported or that are specified as being ignored when not understood.
6. Determine whether the JWT is a JWS or a JWE using any of the methods described in Section 9 of [JWE].
7. Depending upon whether the JWT is a JWS or JWE, there are two cases:
    - If the JWT is a JWS, follow the steps specified in [JWS] for validating a JWS. Let the Message be the result of base64url decoding the JWS Payload.
    - Else, if the JWT is a JWE, follow the steps specified in [JWE] for validating a JWE. Let the Message be the resulting plaintext.
8. If the JOSE Header contains a `cty` (content type) value of `JWT`, then the Message is a JWT that was the subject of nested signing or encryption operations. In this case, return to Step 1, using the Message as the JWT.
9. Otherwise, base64url decode the Message following the restriction that no line breaks, whitespace, or other additional characters have been used.
10. Verify that the resulting octet sequence is a UTF-8-encoded representation of a completely valid JSON object conforming to RFC 7159 [RFC7159]; let the JWT Claims Set be this JSON object.

Finally, note that it is an application decision which algorithms may be used in a given context. Even if a JWT can be successfully validated, unless the algorithms used in the JWT are acceptable to the application, it SHOULD reject the JWT.

## 7.3. String Comparison Rules

Processing a JWT inevitably requires comparing known strings to members and values in JSON objects. For example, in checking what the algorithm is, the Unicode [UNICODE] string encoding `alg` will be checked against the member names in the JOSE Header to see if there is a matching Header Parameter name.

The JSON rules for doing member name comparison are described in Section 8.3 of RFC 7159 [RFC7159]. Since the only string comparison operations that are performed are equality and inequality, the same rules can be used for comparing both member names and member values against known strings.

These comparison rules MUST be used for all JSON string comparisons except in cases where the definition of the member explicitly calls out that a different comparison rule is to be used for that member value. In this specification, only the `typ` and `cty` member values do not use these comparison rules.

Some applications may include case-insensitive information in a case-sensitive value, such as including a DNS name as part of the `iss` (issuer) claim value. In those cases, the application may need to define a convention for the canonical case to use for representing the case-insensitive portions, such as lowercasing them, if more than one party might need to produce the same value so that they can be compared. (However, if all other parties consume whatever value the producing party emitted verbatim without attempting to compare it to an independently produced value, then the case used by the producer will not matter.)

## 8. Implementation Requirements

This section defines which algorithms and features of this specification are mandatory to implement. Applications using this specification can impose additional requirements upon implementations that they use. For instance, one application might require support for encrypted JWTs and Nested JWTs, while another might require support for signing JWTs with the Elliptic Curve Digital Signature Algorithm (ECDSA) using the P-256 curve and the SHA-256 hash algorithm (`ES256`).

Of the signature and MAC algorithms specified in JSON Web Algorithms [JWA], only HMAC SHA-256 (`HS256`) and `none` MUST be implemented by conforming JWT implementations. It is RECOMMENDED that implementations also support RSASSA-PKCS1-v1_5 with the SHA-256 hash algorithm (`RS256`) and ECDSA using the P-256 curve and the SHA-256 hash algorithm (`ES256`). Support for other algorithms and key sizes is OPTIONAL.

Support for encrypted JWTs is OPTIONAL. If an implementation provides encryption capabilities, of the encryption algorithms specified in [JWA], only RSAES-PKCS1-v1_5 with 2048-bit keys (`RSA1_5`), AES Key Wrap with 128- and 256-bit keys (`A128KW` and `A256KW`), and the composite authenticated encryption algorithm using AES-CBC and HMAC SHA-2 (`A128CBC-HS256` and `A256CBC-HS512`) MUST be implemented by conforming implementations. It is RECOMMENDED that implementations also support using Elliptic Curve Diffie-Hellman Ephemeral Static (ECDH-ES) to agree upon a key used to wrap the Content Encryption Key (`ECDH-ES+A128KW` and `ECDH-ES+A256KW`) and AES in Galois/Counter Mode (GCM) with 128- and 256-bit keys (`A128GCM` and `A256GCM`). Support for other algorithms and key sizes is OPTIONAL.

Support for Nested JWTs is OPTIONAL.

## 9. URI for Declaring that Content is a JWT

This specification registers the URN `urn:ietf:params:oauth:token-type:jwt` for use by applications that declare content types using URIs (rather than, for instance, media types) to indicate that the content referred to is a JWT.

## 10. IANA Considerations

### 10.1. JSON Web Token Claims Registry

This section establishes the IANA "JSON Web Token Claims" registry for JWT Claim Names. The registry records the Claim Name and a reference to the specification that defines it. This section registers the Claim Names defined in Section 4.1.

Values are registered on a Specification Required [RFC5226] basis after a three-week review period on the jwt-reg-review@ietf.org mailing list, on the advice of one or more Designated

Experts. However, to allow for the allocation of values prior to publication, the Designated Experts may approve registration once they are satisfied that such a specification will be published.

Registration requests sent to the mailing list for review should use an appropriate subject (e.g., "Request to register claim: example").

Within the review period, the Designated Experts will either approve or deny the registration request, communicating this decision to the review list and IANA. Denials should include an explanation and, if applicable, suggestions as to how to make the request successful. Registration requests that are undetermined for a period longer than 21 days can be brought to the IESG's attention (using the iesg@ietf.org mailing list) for resolution.

Criteria that should be applied by the Designated Experts includes determining whether the proposed registration duplicates existing functionality, whether it is likely to be of general applicability or whether it is useful only for a single application, and whether the registration description is clear.

IANA must only accept registry updates from the Designated Experts and should direct all requests for registration to the review mailing list.

It is suggested that multiple Designated Experts be appointed who are able to represent the perspectives of different applications using this specification, in order to enable broadly informed review of registration decisions. In cases where a registration decision could be perceived as creating a conflict of interest for a particular Expert, that Expert should defer to the judgment of the other Experts.

### 10.1.1. Registration Template

Claim Name:

> The name requested (e.g., "iss"). Because a core goal of this specification is for the resulting representations to be compact, it is RECOMMENDED that the name be short -- that is, not to exceed 8 characters without a compelling reason to do so. This name is case sensitive. Names may not match other registered names in a case-insensitive manner unless the Designated Experts state that there is a compelling reason to allow an exception.

Claim Description:

> Brief description of the claim (e.g., "Issuer").

Change Controller:

> For Standards Track RFCs, list the "IESG". For others, give the name of the responsible party. Other details (e.g., postal address, email address, home page URI) may also be included.

Specification Document(s):

> Reference to the document or documents that specify the parameter, preferably including URIs that can be used to retrieve copies of the documents. An indication of the relevant sections may also be included but is not required.

### 10.1.2. Initial Registry Contents

- Claim Name: `iss`
- Claim Description: Issuer
- Change Controller: IESG
- Specification Document(s): Section 4.1.1 of RFC 7519

- Claim Name: `sub`
- Claim Description: Subject
- Change Controller: IESG
- Specification Document(s): Section 4.1.2 of RFC 7519

- Claim Name: `aud`

- Claim Description: Audience
- Change Controller: IESG
- Specification Document(s): Section 4.1.3 of RFC 7519

- Claim Name: `exp`
- Claim Description: Expiration Time
- Change Controller: IESG
- Specification Document(s): Section 4.1.4 of RFC 7519

- Claim Name: `nbf`
- Claim Description: Not Before
- Change Controller: IESG
- Specification Document(s): Section 4.1.5 of RFC 7519

- Claim Name: `iat`
- Claim Description: Issued At
- Change Controller: IESG
- Specification Document(s): Section 4.1.6 of RFC 7519

- Claim Name: `jti`
- Claim Description: JWT ID
- Change Controller: IESG
- Specification Document(s): Section 4.1.7 of RFC 7519

## 10.2. Sub-Namespace Registration of urn:ietf:params:oauth:token-type:jwt

### 10.2.1. Registry Contents

This section registers the value `token-type:jwt` in the IANA "OAuth URI" registry established by "An IETF URN Sub-Namespace for OAuth" [RFC6755], which can be used to indicate that the content is a JWT.

- URN: urn:ietf:params:oauth:token-type:jwt
- Common Name: JSON Web Token (JWT) Token Type
- Change Controller: IESG
- Specification Document(s): RFC 7519

## 10.3. Media Type Registration

### 10.3.1. Registry Contents

This section registers the `application/jwt` media type [RFC2046] in the "Media Types" registry [IANA.MediaTypes] in the manner described in RFC 6838 [RFC6838], which can be used to indicate that the content is a JWT.

- Type name: application
- Subtype name: jwt
- Required parameters: n/a
- Optional parameters: n/a
- Encoding considerations: 8bit; JWT values are encoded as a series of base64url-encoded values (some of which may be the empty string) separated by period ('.') characters.
- Security considerations: See the Security Considerations section of RFC 7519
- Interoperability considerations: n/a
- Published specification: RFC 7519
- Applications that use this media type: OpenID Connect, Mozilla Persona, Salesforce, Google, Android, Windows Azure, Amazon Web Services, and numerous others
- Fragment identifier considerations: n/a
- Additional information:

  Magic number(s): n/a

  File extension(s): n/a

  Macintosh file type code(s): n/a

- Person & email address to contact for further information:
  Michael B. Jones, mbj@microsoft.com
- Intended usage: COMMON
- Restrictions on usage: none
- Author: Michael B. Jones, mbj@microsoft.com
- Change controller: IESG
- Provisional registration? No

## 10.4. Header Parameter Names Registration

This section registers specific Claim Names defined in Section 4.1 in the IANA "JSON Web Signature and Encryption Header Parameters" registry established by [JWS] for use by claims replicated as Header Parameters in JWEs, per Section 5.3.

### 10.4.1. Registry Contents

- Header Parameter Name: `iss`
- Header Parameter Description: Issuer
- Header Parameter Usage Location(s): JWE
- Change Controller: IESG
- Specification Document(s): Section 4.1.1 of RFC 7519

- Header Parameter Name: `sub`
- Header Parameter Description: Subject
- Header Parameter Usage Location(s): JWE
- Change Controller: IESG
- Specification Document(s): Section 4.1.2 of RFC 7519

- Header Parameter Name: `aud`
- Header Parameter Description: Audience
- Header Parameter Usage Location(s): JWE
- Change Controller: IESG
- Specification Document(s): Section 4.1.3 of RFC 7519

## 11. Security Considerations

All of the security issues that are pertinent to any cryptographic application must be addressed by JWT/JWS/JWE/JWK agents. Among these issues are protecting the user's asymmetric private and symmetric secret keys and employing countermeasures to various attacks.

All the security considerations in the JWS specification also apply to JWT, as do the JWE security considerations when encryption is employed. In particular, Sections 10.12 ("JSON Security Considerations") and 10.13 ("Unicode Comparison Security Considerations") of [JWS] apply equally to the JWT Claims Set in the same manner that they do to the JOSE Header.

## 11.1. Trust Decisions

The contents of a JWT cannot be relied upon in a trust decision unless its contents have been cryptographically secured and bound to the context necessary for the trust decision. In particular, the key(s) used to sign and/or encrypt the JWT will typically need to verifiably be under the control of the party identified as the issuer of the JWT.

## 11.2. Signing and Encryption Order

While syntactically the signing and encryption operations for Nested JWTs may be applied in any order, if both signing and encryption are necessary, normally producers should sign the message and then encrypt the result (thus encrypting the signature). This prevents attacks in which the signature is stripped, leaving just an encrypted message, as well as providing privacy for the signer. Furthermore, signatures over encrypted text are not considered valid in many jurisdictions.

Note that potential concerns about security issues related to the order of signing and encryption operations are already addressed by the underlying JWS and JWE specifications; in particular, because JWE only supports the use of authenticated encryption algorithms, cryptographic concerns about the potential need to sign after encryption that apply in many contexts do not apply to this specification.

## 12. Privacy Considerations

A JWT may contain privacy-sensitive information. When this is the case, measures MUST be taken to prevent disclosure of this information to unintended parties. One way to achieve this is to use an encrypted JWT and authenticate the recipient. Another way is to ensure that JWTs containing unencrypted privacy-sensitive information are only transmitted using protocols utilizing encryption that support endpoint authentication, such as Transport Layer Security (TLS). Omitting privacy-sensitive information from a JWT is the simplest way of minimizing privacy issues.

## 13. References

### 13.1. Normative References

| | |
|---|---|
| **[ECMAScript]** | Ecma International, "ECMAScript Language Specification, 5.1 Edition", ECMA Standard 262, June 2011. |
| **[IANA.MediaTypes]** | IANA, "Media Types" |
| **[JWA]** | Jones, M., "JSON Web Algorithms (JWA)", RFC 7518, May 2015. |
| **[JWE]** | Jones, M. and J. Hildebrand, "JSON Web Encryption (JWE)", RFC 7516, May 2015. |
| **[JWS]** | Jones, M., Bradley, J. and N. Sakimura, "JSON Web Signature (JWS)", RFC 7515, May 2015. |
| **[RFC20]** | Cerf, V., "ASCII format for Network Interchange", STD 80, RFC 20, October 1969. |
| **[RFC2046]** | Freed, N. and N. Borenstein, "Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types", RFC 2046, November 1996. |
| **[RFC2119]** | Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997. |
| **[RFC3986]** | Berners-Lee, T., Fielding, R. and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax", STD 66, RFC 3986, January 2005. |
| **[RFC4949]** | Shirey, R., "Internet Security Glossary, Version 2", RFC 4949, August 2007. |
| **[RFC7159]** | Bray, T., "The JavaScript Object Notation (JSON) Data Interchange Format", RFC 7159, March 2014. |
| **[UNICODE]** | The Unicode Consortium, "The Unicode Standard" |

### 13.2. Informative References

| | |
|---|---|
| **[CanvasApp]** | Facebook, "Canvas Applications", 2010. |
| **[JSS]** | Bradley, J. and N. Sakimura (editor), "JSON Simple Sign", September 2010. |
| **[MagicSignatures]** | Panzer, J., Laurie, B. and D. Balfanz, "Magic Signatures", January 2011. |
| **[OASIS.saml-core-2.0-os]** | Cantor, S., Kemp, J., Philpott, R. and E. Maler, "Assertions and Protocols for the OASIS Security Assertion Markup Language (SAML) V2.0", OASIS Standard saml-core-2.0-os, March 2005. |
| **[POSIX.1]** | IEEE, "The Open Group Base Specifications Issue 7", IEEE Std 1003.1, 2013 Edition, 2013. |
| **[RFC3275]** | Eastlake, D., Reagle, J. and D. Solo, "(Extensible Markup Language) XML-Signature Syntax and Processing", RFC 3275, March 2002. |
| **[RFC3339]** | Klyne, G. and C. Newman, "Date and Time on the Internet: Timestamps", RFC 3339, July 2002. |

| | |
|---|---|
| **[RFC4122]** | Leach, P., Mealling, M. and R. Salz, "A Universally Unique IDentifier (UUID) URN Namespace", RFC 4122, July 2005. |
| **[RFC5226]** | Narten, T. and H. Alvestrand, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 5226, May 2008. |
| **[RFC6755]** | Campbell, B. and H. Tschofenig, "An IETF URN Sub-Namespace for OAuth", RFC 6755, October 2012. |
| **[RFC6838]** | Freed, N., Klensin, J. and T. Hansen, "Media Type Specifications and Registration Procedures", BCP 13, RFC 6838, January 2013. |
| **[SWT]** | Hardt, D. and Y. Goland, "Simple Web Token (SWT)", Version 0.9.5.1, November 2009. |
| **[W3C.CR-xml11-20060816]** | Cowan, J., "Extensible Markup Language (XML) 1.1 (Second Edition)", World Wide Web Consortium Recommendation REC-xml11-20060816, August 2006. |
| **[W3C.REC-xml-c14n-20010315]** | Boyer, J., "Canonical XML Version 1.0", World Wide Web Consortium Recommendation REC-xml-c14n-20010315, March 2001. |

## Appendix A. JWT Examples

This section contains examples of JWTs. For other example JWTs, see Section 6.1 of this document and Appendices A.1 - A.3 of [JWS].

## A.1. Example Encrypted JWT

This example encrypts the same claims as used in Section 3.1 to the recipient using RSAES-PKCS1-v1_5 and AES_128_CBC_HMAC_SHA_256.

The following example JOSE Header declares that:

- The Content Encryption Key is encrypted to the recipient using the RSAES-PKCS1-v1_5 algorithm to produce the JWE Encrypted Key.
- Authenticated encryption is performed on the plaintext using the AES_128_CBC_HMAC_SHA_256 algorithm to produce the JWE Ciphertext and the JWE Authentication Tag.

```
{"alg":"RSA1_5","enc":"A128CBC-HS256"}
```

Other than using the octets of the UTF-8 representation of the JWT Claims Set from Section 3.1 as the plaintext value, the computation of this JWT is identical to the computation of the JWE in Appendix A.2 of [JWE], including the keys used.

The final result in this example (with line breaks for display purposes only) is:

```
eyJhbGciOiJSU0ExXzUiLCJlbmMiOiJBMTI4Q0JDLUhTMjU2In0.
QR1Owv2ug2WyPBnbQrRARTeEk9kDO2w8qDcjiHnSJflSdv1iNqhWXaKH4MqAkQtM
oNfABIPJaZm0HaA415sv3aeuBWnD8J-Ui7Ah6cWafs3ZwwFKDFUUsWHSK-IPKxLG
TkND09XyjORj_CHAgOPJ-Sd8ONQRnJvWn_hXV1BNMHzUjPyYwEsRhDhzjAD26ima
sOTsgruobpYGoQcXUwFDn7moXPRfDE8-NoQX7N7ZYMmpUDkR-Cx9obNGwJQ3nM52
YCitxoQVPzjbl7WBuB7AohdBoZOdZ24WlN1lVIeh8v1K4krB8xgKvRU8kgFrEn_a
1rZgN5TiysnmzTROF869lQ.
AxY8DCtDaGlsbGljb3RoZQ.
MKOle7UQrG6nSxTLX6Mqwt0orbHvAKeWnDYvpIAeZ72deHxz3roJDXQyhxx0wKaM
HDjUEOKIwrtkHthpqEanSBNYHZgmNOV7sln1Eu9g3J8.
fiK51VwhsxJ-siBMR-YFiA
```

## A.2. Example Nested JWT

This example shows how a JWT can be used as the payload of a JWE or JWS to create a Nested JWT. In this case, the JWT Claims Set is first signed, and then encrypted.

The inner signed JWT is identical to the example in Appendix A.2 of [JWS]. Therefore, its computation is not repeated here. This example then encrypts this inner JWT to the recipient using RSAES-PKCS1-v1_5 and AES_128_CBC_HMAC_SHA_256.

The following example JOSE Header declares that:

- The Content Encryption Key is encrypted to the recipient using the RSAES-PKCS1-v1_5 algorithm to produce the JWE Encrypted Key.
- Authenticated encryption is performed on the plaintext using the AES_128_CBC_HMAC_SHA_256 algorithm to produce the JWE Ciphertext and the JWE Authentication Tag.
- The plaintext is itself a JWT.

```
{"alg":"RSA1_5","enc":"A128CBC-HS256","cty":"JWT"}
```

Base64url encoding the octets of the UTF-8 representation of the JOSE Header yields this encoded JOSE Header value:

```
eyJhbGciOiJSU0ExXzUiLCJlbmMiOiJBMTI4Q0JDLUhTMjU2IiwiY3R5IjoiSldUIn0
```

The computation of this JWT is identical to the computation of the JWE in Appendix A.2 of [JWE], other than that different JOSE Header, plaintext, JWE Initialization Vector, and Content Encryption Key values are used. (The RSA key used is the same.)

The plaintext used is the octets of the ASCII [RFC20] representation of the JWT at the end of Appendix A.2.1 of [JWS] (with all whitespace and line breaks removed), which is a sequence of 458 octets.

The JWE Initialization Vector value used (using JSON array notation) is:

[82, 101, 100, 109, 111, 110, 100, 32, 87, 65, 32, 57, 56, 48, 53, 50]

This example uses the Content Encryption Key represented by the base64url-encoded value below:

```
GawgguFyGrWKav7AX4VKUg
```

The final result for this Nested JWT (with line breaks for display purposes only) is:

```
eyJhbGciOiJSU0ExXzUiLCJlbmMiOiJBMTI4Q0JDLUhTMjU2IiwiY3R5IjoiSldU
In0.
g_hEwksO1Ax8Qn7HoN-BVeBoa8FXe0kpyk_XdcSmxvcM5_P296JXXtoHISr_DD_M
qewaQSH4dZOQHoUgKLeFly-9RI11TG-_Ge1bZFazBPwKC5lJ6OLANLMd0QSL4fYE
b9ERe-epKYE3xb2jfY1AltHqBO-PM6j23Guj2yDKnFv6WO72tteVzm_2n17SBFvh
DuR9a2nHTE67pe0XGBUS_TK7ecA-iVq5COeVdJR4U4VZGGlxRGPLRHvolVLEHx6D
YyLpw30Ay9R6d68YCLi9FYTq3hIXPK_-dmPlOUlKvPr1GgJzRoeC9G5qCvdcHWsq
JGTO_z3Wfo5zsqwkxruxwA.
UmVkbW9uZCBXQSA5ODA1Mg.
VwHERHPvCNcHHpTjkoigx3_ExK0Qc71RMEParpatm0X_qpg-w8kozSjfNIPPXiTB
BLXR65CIPkFqz4l1Ae9w_uowKiwyi9acgVztAi-pSL8GQSXnaamh9kX1mdh3M_TT
-FZGQFQsFhu0Z72gJKGdfGE-OE7hS1zuBD5oEUfk0Dmb0VzWEzpxxiSSBbBAzP10
l56pPfAtrjEYw-7ygeMkwBl6Z_mLS6w6xUgKlvW6ULmkV-uLC4FUiyKECK4e3WZY
Kw1bpgIqGYsw2v_grHjszJZ-_I5uM-9RA8ycX9KqPRp9gc6pXmoU_-27ATs9XCvr
ZXUtK2902AUzqpeEUJYjWWxSNsS-r1TJ1I-FMJ4XyAiGrfmo9hQPcNBYxPz3GQb2
8Y5CLSQfNgKSGt0A4isp1hBUXBHAndgtcslt7ZoQJaKe_nNJgNliWtWpJ_ebuOpE
l8jdhehdccnRMIwAmU1n7SPkmhIl1HlSOpvcvDfhUN5wuqU955vOBvfkBOh5A11U
zBuo2WlgZ6hYi9-e3w29bR0C2-pp3jbqxEDw3iWaf2dc5b-LnR0FEYXvI_tYk5rd
_J9N0mg0tQ6RbpxNEMNoA9QWk5lgdPvbh9BaO195abQ.
AVO9iT5AV4CzvDJCdhSFlQ
```

## Appendix B. Relationship of JWTs to SAML Assertions

Security Assertion Markup Language (SAML) 2.0 [OASIS.saml-core-2.0-os] provides a standard for creating security tokens with greater expressivity and more security options than supported by JWTs. However, the cost of this flexibility and expressiveness is both size and complexity. SAML's use of XML [W3C.CR-xml11-20060816] and XML Digital Signature (DSIG) [RFC3275] contributes to the size of SAML Assertions; its use of XML and especially XML Canonicalization [W3C.REC-xml-c14n-20010315] contributes to their complexity.

JWTs are intended to provide a simple security 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 [RFC7159] object encoding syntax. It also supports securing tokens using Message Authentication Codes (MACs) and digital signatures using a smaller (and less flexible) format than XML DSIG.

Therefore, while JWTs can do some of the things SAML Assertions do, JWTs are not intended as a full replacement for SAML Assertions, but rather as a token format to be used when ease of implementation or compactness are considerations.

SAML Assertions are always statements made by an entity about a subject. JWTs are often used in the same manner, with the entity making the statements being represented by the `iss` (issuer) claim, and the subject being represented by the `sub` (subject) claim. However, with these claims being optional, other uses of the JWT format are also permitted.

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

Both JWTs and SWTs [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 signature, MAC, and encryption algorithms.

## Acknowledgements

The authors acknowledge that the design of JWTs was intentionally influenced by the design and simplicity of SWTs [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 document.

This specification is the work of the OAuth working group, which includes dozens of active and dedicated participants. In particular, the following individuals contributed ideas, feedback, and wording that influenced this specification:

Dirk Balfanz, Richard Barnes, Brian Campbell, Alissa Cooper, Breno de Medeiros, Stephen Farrell, Yaron Y. Goland, Dick Hardt, Joe Hildebrand, Jeff Hodges, Edmund Jay, Warren Kumari, Ben Laurie, Barry Leiba, Ted Lemon, James Manger, Prateek Mishra, Kathleen Moriarty, Tony Nadalin, Axel Nennker, John Panzer, Emmanuel Raviart, David Recordon, Eric Rescorla, Jim Schaad, Paul Tarjan, Hannes Tschofenig, Sean Turner, and Tom Yu.

Hannes Tschofenig and Derek Atkins chaired the OAuth working group and Sean Turner, Stephen Farrell, and Kathleen Moriarty served as Security Area Directors during the creation of this specification.

## Authors' Addresses

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

**John Bradley**
Ping Identity
EMail: ve7jtb@ve7jtb.com
URI: http://www.thread-safe.com/

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

URI: http://nat.sakimura.org/