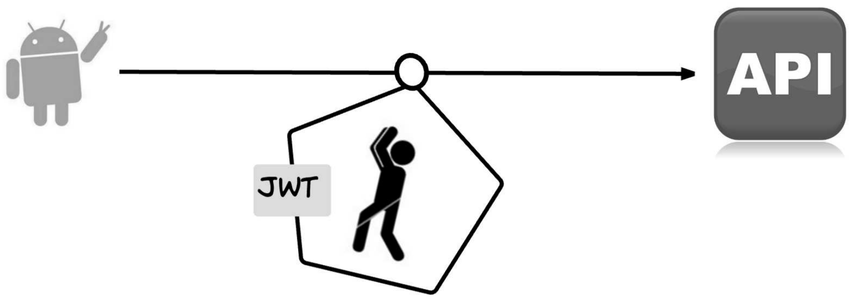
Images haven't loaded yet. Please exit printing, wait for images to load, and try to print again.

Apr 27, 2016 · 22 min read

JWT, JWS and JWE for Not So Dummies! (Part I)

JSON Web Token (JWT) defines a container to transport data between interested parties. It became an IETF standard in May 2015 with the RFC 7519. There are multiple applications of JWT. The OpenID Connect is one of them. In OpenID Connect the *id_token* is represented as a JWT. Both in securing APIs and Microservices, the JWT is used as a way to propagate and verify end-user identity.



Propagating user identity from a mobile app to a secured API. The API has to verify and accept the JWT prior to any further processing.

A JWT can be used to:

- Propagate one's identity between interested parties.
- Propagate user entitlements between interested parties.
- Transfer data securely between interested parties over a unsecured channel.
- Assert one's identity, given that the recipient of the JWT trusts the asserting party.

Following is a sample JWT, which is returned back from the Google OpenID Connect provider. Here the Google, which is the identity provider, asserts the identity of an end-user and passes the JWT to a service provider (a web or native mobile application).

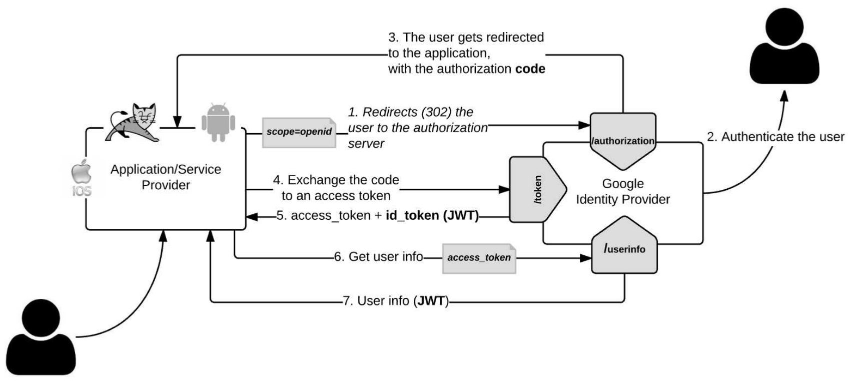
> *eyJhbGciOiJSUzI1NiIsImtpZCI6Ijc4YjRjZjIzNjU2ZGMzOTUzNjRmMWI2Y*
> *zAyOTA3NjkxZjJjZGZmZTEifQ.* ***eyJpc3MiOiJhY2NvdW50cy5nb29nbG***
> ***UuY29tIiwic3ViIjoiMTEwNTAyMjUxMTU4OTIwMTQ3NzMyIiwiYXp***
> ***wIjoiODI1MjQ5ODM1NjU5LXRlOHFnbDcwMWtnb25ub21ucDRzcX***
> ***Y3ZXJodTEyMTFzLmFwcHMuZ29vZ2xldXNlcmNvbnRlbnQuY29tIiw***
> ***iZW1haWwiOiJwcmFiYXRoQHdzbzIuY29tIiwiYXRfaGFzaCI6InpmO***
> ***DZ2TnVsc0xCOGdGYXFSd2R6WWciLCJlbWFpbF92ZXJpZmllZCI6dH***
> ***J1ZSwiYXVkIjoiODI1MjQ5ODM1NjU5LXRlOHFnbDcwMWtnb25ub2***
> ***1ucDRzcXY3ZXJodTEyMTFzLmFwcHMuZ29vZ2xldXNlcmNvbnRlbn***
> ***QuY29tIiwiaGQiOiJ3c28yLmNvbSIsImlhdCI6MTQwMTkwODI3MS***
> ***wiZXhwIjoxNDAxOTEyMTcxfQ*** *.TVKv-pdyvk2gW8sGsCbsnkqsrS0T-*
> *H00xnY6ETkIfgIxfotvFn5IwKm3xyBMpy0FFe0Rb5Ht8AEJV6PdWyxz8rM*
> *gX2HROWqSo_RfEfUpBb4iOsq4W28KftW5H0IA44VmNZ6zU4YTqPSt4T*
> *PhyFC9fP2D_Hg7JQozpQRUfbWTJI*

This looks gibberish till you break it by periods (.) and base64url-decode each part. There are two periods in it, which break the whole string into three parts. Once you base64url-decode the fist part, it appears like below:

> *{"alg":"RS256","kid":"78b4cf23656dc395364f1b6c02907691f2cdffe1"}*

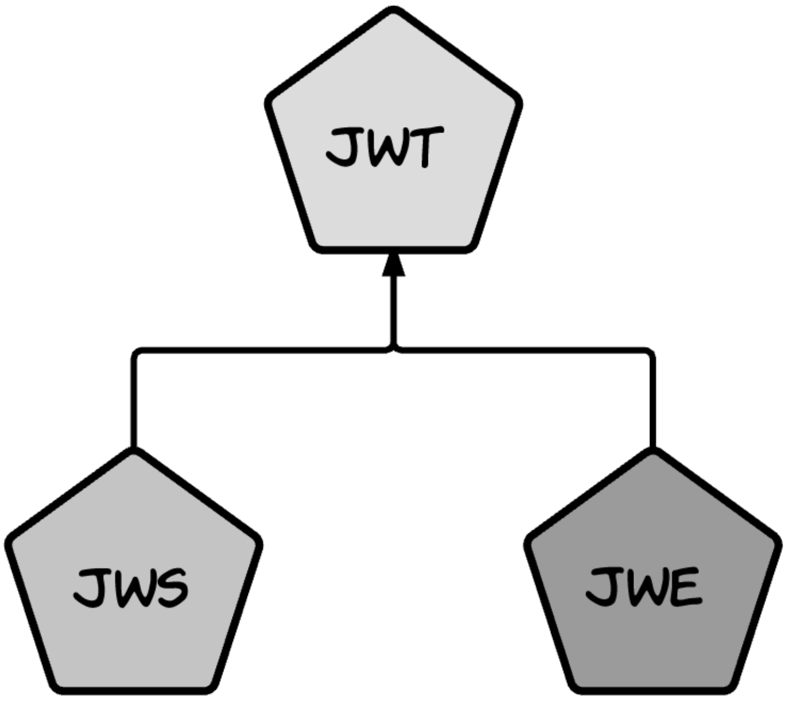
JOSE Header

This first part(once parted by the periods) of the JWT is known as the JOSE header . JOSE stands for *Javascript Object Signing and Encryption* —and it's the name of the IETF working group, which works on standardizing the representation of integrity-protected data using JSON data structures. The above JOSE header indicates that it's a signed message. Google asserts the identity of the end-user by signing the JWT, which carries data related to the user's identity.


3. The user gets redirected to the application, with the authorization **code**
1. Redirects (302) the user to the authorization server
scope=openid
/authorization
2. Authenticate the user
Application/Service Provider
iOS
4. Exchange the code to an access token
5. access_token + **id_token (JWT)**
/token
Google Identity Provider
/userinfo
6. Get user info
access_token
7. User info (**JWT**)

Login to a service provider via Google OpenID Connect. In step-5 and step-7 user information is returned back to the service provider in a JWT, signed by Google.

A signed JWT is known as a JWS (JSON Web Signature). In fact a JWT does not exist itself—either it has to be a JWS or a JWE (JSON Web Encryption). Its like an abstract class—the JWS and JWE are the concrete implementations.



Going back to the JOSE header returned back from Google, both the ***alg*** and ***kid*** elements there, are not defined in the JWT specification, but in the JSON Web Signature (JWS) specification. The JWT specification only defines two elements (***typ*** and ***cty***) in the JOSE header and both the JWS and JWE specifications extend it to add more appropriate elements.

> ***typ (type)****: The* ***typ*** *element is used to define the media type of the complete JWT. A media type is an identifier, which defines the format of the content, transmitted on the Internet. There are two types of components that process a JWT: JWT implementations and JWT applications. Nimbus is a JWT implementation in Java. The Nimbus library knows how to build and parse a JWT. A JWT application can be anything, which uses JWTs internally. A JWT application uses a JWT implementation to build or parse a JWT. In this case, the* ***typ*** *element is just another element for the JWT implementation. It will not try to interpret the value of it, but the JWT application would. The* ***typ*** *element helps JWT applications to differentiate the content of the JWT when the values that are not JWTs could also be present in an application data structure along with a JWT object. This is an optional element and if present for a JWT, it is recommended to use* ***JWT*** *as the media type.*

> ***cty (content type)****: The* ***cty*** *element is used to define the structural information about the JWT. It is only recommended to use this element in the case of a nested JWT.*

The JWS specification is not bound to any specific algorithm. All applicable algorithms for signing are defined under the JSON Web Algorithms (JWA) specification, which is the RFC 7518. The section 3.1 of RFC 7518 defines all possible ***alg*** element values for a JWS token. The value of the ***kid*** element provides an indication or a hint about the key, which is used to sign the message. Looking at the ***kid***, the recipient of the message should know where and how to lookup for the key and find it.

> *In a JWT, 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. Both under the JWS and JWE, the JOSE header is a must—or in other words there exists no JWT without a JOSE header.*

Unsecured JWT

A JWT can be either a JWS or a JWE object. Unsecured JWT is a JWS object where in the JOSE header the value of the ***alg*** element is set to ***none***. In other words, an unsecured JWT is a JWS without a signature (sounds bit weird though :-)). When propagating user identity and entitlement information over an unsecured JWT, it is expected that the underlying transport will provide a guarantee on the integrity and the confidentiality of the token; use TLS.

Claim Set

Focus back on the sample JWT returned back from Google. More precisely now we know its a JWS. Following shows the base64url-decoded claim set returned back from Google. The second part of the JWT (when parted by the period (.)) is known as the JWT claim set. White-spaces can be explicitly retained while building the JWT claim set—no canonicalization is required before base64url-encoding or decoding. *Canonicalization* is the process of converting different forms of a message into a single standard form. This is used mostly before signing XML messages.

```
{ ⊟
   "iss":"accounts.google.com",
   "sub":"110502251158920147732",
   "azp":"825249835659-np4sqv7erhu1211s.apps.googleusercontent.com",
   "email":"prabath@wso2.com",
   "at_hash":"zf86vNulsLB8gFaqRwdzYg",
   "email_verified":true,
   "aud":"825249835659-np4sqv7erhu1211s.apps.googleusercontent.com",
   "hd":"wso2.com",
   "iat":1401908271,
   "exp":1401912171
}
```

The JWT claim set represents a JSON object whose members are the claims asserted by the JWT issuer. Each claim name within a JWT must be unique. If there are duplicate claim names, then the JWT parser could either return a parsing error or just return back the claims set with the very last duplicate claim. JWT specification does not explicitly define what claims are mandatory and what are optional. It's up to the each application of JWT to define mandatory and optional claims. For example, the OpenID Connect specification defines the mandatory and optional claims. According to the OpenID Connect core specification, ***iss***, ***sub***, ***aud***, ***exp*** and ***iat*** are treated as mandatory elements, while ***auth_time***, ***nonce***, ***acr***, ***amr*** and ***azp*** are optional elements. In addition to the mandatory and optional claims, which are defined in the specification, the identity provider can include additional elements into the JWT claim set.

Signature

Once again, focus back on the sample JWT returned back from Google. The third part of the JWT (when parted by the period (.)), is the signature, which is also base64url-encoded. The cryptographic elements related to the signature are defined in the JOSE header. In this particular example, Google uses RSASSA-PKCS1-V1_5 with the SHA-256 hashing algorithm, which is expressed by the value of the ***alg*** element in the JOSE header: ***RS256***.

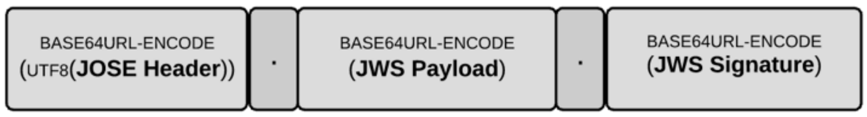
Serialization

A signed or an encrypted message can be serialized in two ways by following the JWS or JWE specification: the *JWS/JWE compact serialization* and the *JWS/JWE JSON serialization*. The Google OpenID Connect response discussed before uses the JWS compact serialization. In fact, the OpenID Connect specification mandates to use JWS compact serialization and JWE compact serialization whenever necessary.

Now we can further refine our definition of the JWT. So far we know that both the JWS and JWE tokens are instances of the JWT. But that is not 100% precise. We call a JWS or JWE, a JWT only if it follows the compact serialization. Any JWT must follow compact serialization. In other words a JWS or JWE token, which follows JSON serialization cannot be called as a JWT.

JWS Compact Serialization

JWS compact serialization represents a signed JWT as a compact URL-safe string. This compact string has three main elements separated by periods (.): ***the JOSE header***, ***the JWS payload*** and ***the JWS signature***. If you use compact serialization against a JSON payload (or any payload—even XML), then you can have only a single signature, which is computed over the complete ***JOSE header*** and ***JWS payload***.


BASE64URL-ENCODE (UTF8(JOSE Header))
.
BASE64URL-ENCODE (JWS Payload)
.
BASE64URL-ENCODE (JWS Signature)

The structure of a JWS token formed by JWS compact serialization.

JWS Compact Serialization—Signing Process

Following lists out the signing process of a JWS under the compact serialization.

- Build a JSON object including all the header elements, which express the cryptographic properties of the JWS token—this is known as the JOSE header. As discussed before, the token issuer should advertise in the JOSE header, the public key corresponding to the key used to sign the message. This can be expressed via any of these header elements: ***jku***, ***jwk***, ***kid***, ***x5u***, ***x5c***, ***x5t*** and ***x5t#s256***.

- Compute the base64url-encoded value against the UTF-8 encoded JOSE header from the 1st step, to produce the 1st element of the JWS token.

- Construct the payload or the content to be signed—this is known as the JWS payload. **The payload is not necessarily JSON**—it can be any content. Yes, you read it correctly, the payload of a JWS necessarily need not to be JSON - if you'd like it can be XML too.

- Compute the base64url-encoded value of the JWS payload from the previous step to produce the 2nd element of the JWS token.

- Build the message to compute the digital signature or the Mac. The message is constructed as ASCII(BASE64URL-ENCODE(UTF8(JOSE Header)) '.' BASE64URL-ENCODE(JWS Payload)).

- Compute the signature over the message constructed in the previous step, following the signature algorithm defined by the JOSE header element ***alg***. The message is signed using the private key corresponding to the public key advertised in the JOSE header.

- Compute the base64url encoded value of the JWS signature produced in the previous step, which is the 3rd element of the serialized JWS token.

- Now we have all the elements to build the JWS token in the following manner. The line breaks are introduced only for clarity.

> *BASE64URL(UTF8(JWS Protected Header)) '.'*

> *BASE64URL(JWS Payload) '.'*

> *BASE64URL(JWS Signature)*

JWS JSON Serialization

In contrast to the JWS compact serialization, the JWS JSON serialization can produce multiple signatures over the same JWS payload along with multiple JOSE headers. The ultimate serialized form under JWS JSON serialization wraps the signed payload in a JSON object, with all the related metadata. This JSON object includes 2 top-level elements: ***payload*** and ***signatures*** *(which is a JSON array)*, and three sub elements under each entry of the ***signatures*** array: ***protected***, ***header*** and ***signature***.

Following is an example of a JWS token, which is serialized under JWS JSON serialization.

```
{
   "payload":"eyJpc3MiOiJqb2UiLA0KICJleHAiOjEzMDA4MTkzOD",
   "signatures":[
      {
         "protected":"eyJhbGciOiJSUzI1NiJ9",
         "header":{
            "kid":"2014-06-29"
         },
         "signature":"cC4hiUPoj9Eetdgtv3hF80EGrhuB"
      },
      {
         "protected":"eyJhbGciOiJFUzI1NiJ9",
         "header":{
            "kid":"e909097a-ce81-4036-9562-d21d2992db0d"
         },
         "signature":"DtEhU3ljbEg8L38VWAfUAqOyKAM"
      }
   ]
}
```

This is neither URL safe nor optimized for compactness. It carries two signatures over the same payload, and each signature and the metadata around it are stored as an element in the JSON array, represented by the ***signatures*** top-level element. Each signature uses a different key to sign, represented by the corresponding ***kid*** header element.

The ***payload*** top-level element of the JSON object includes the base64url-encoded value of the complete JWS payload. The JWS payload not necessarily needs to be a JSON payload, it can be of any content type. The ***payload*** is a required element in the serialized JWS token.

The JWS Protected Header

The JWS protected header is a JSON object that includes the header elements that has to be integrity protected by the signing or MAC algorithm. There can be multiple JWS protected headers in a JWS, serialized under JSON serialization, where each one of them carries the header elements that has to be signed differently. The JSON serialization is useful in selectively signing JOSE header elements, while in contrast JWS compact serialization signs the complete JOSE header.

Each ***protected*** element in the serialized JSON form represents the base64url-encoded value of a JWS protected header. The ***protected*** element is defined under each entry of the ***signatures*** JSON array and includes the base64url-encoded JSON object of header elements, which should be signed. If you base64url-decode the value of the first ***protected*** element in the above code snippet, you will see ***{"alg":"RS256"}***. The ***protected*** element must be present, if there are any JWS protected headers. There can be one ***protected*** element for each entry of the signatures JSON array.

JWS Unprotect Header

The JWS unprotected header is a JSON object, which includes the header elements that are not integrity protected by the signing or MAC algorithm. Each ***header*** element in the serialized JSON form represents the base64url-encoded value of a JWS unprotected header. The ***header*** element is defined under each entry in the ***signatures*** JSON array and includes unprotected header elements corresponding to this signature, which are not signed. Combining both the protected headers and unprotected headers ultimately derives the JOSE header corresponding to this signature (the metadata related to the signature can be either protected or unprotected). In the above code snippet, the complete JOSE header corresponding to the first entry in the ***signatures*** JSON array would be ***{"alg":"RS256", "kid":"2010–12–29"}***, which aggregates both the protected and unprotected headers. The ***header*** element itself is represented as a JSON object and must be present if there are any unprotected header elements. There can be one header element for each entry of the ***signatures*** JSON array.

The ***signatures*** element of the JSON object includes an array of JSON objects, where each element includes a signature or MAC (over the JWS payload and JWS protected header) and the associated metadata. This is a required element. The ***signature*** element, which is inside each entry of the ***signatures*** array carries the base64url-encoded value of the signature computed over the protected header elements (represented by the protected element) and the JWS payload. Both the ***signatures*** and ***signature*** are required elements.

JWS JSON Serialization—Signing Process

Following lists out the signing process of a JWS under the JSON serialization.

- Construct the payload or the content to be signed—this is known as the JWS payload. **The payload is not necessarily JSON**—it can be any content. The ***payload*** element in the serialized JWS token carries the base64url-encoded value of this.

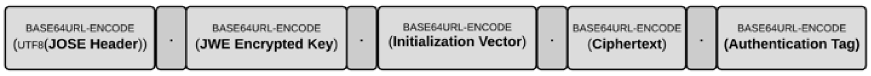
- Decide how many signatures you would need against the payload and for each case which header elements must be signed and which are not.
- Build a JSON object including all the header elements that are to be integrity protected or to be signed. In other words construct the JWS protected header for each signature. The base64url-encoded value of the UTF-8 encoded JWS protected header will produce the value of the corresponding ***protected*** element inside each entry of the ***signatures*** JSON array.
- Build a JSON object including all the header elements that need not to be integrity protected or not to be signed. In other words construct the JWS unprotected header for each signature. This will produce the corresponding ***header*** element inside each entry of the ***signatures*** JSON array.
- Both the JWS protected header and the JWS unprotected header express the cryptographic properties of the corresponding signature—this is known as the JOSE header. As discussed before the token issuer should advertise in the JOSE header, the public key corresponding the key used to sign the message. This can be expressed via any of these header elements: ***jku***, ***jwk***, ***kid***, ***x5u***, ***x5c***, ***x5t*** and ***x5t#s256***.
- Build the message to compute the digital signature or the Mac against each entry in the ***signatures*** JSON array of the serialized JWS token. The message is constructed as ASCII(BASE64URL-ENCODE(UTF8(JWS Protected Header of the corresponding entry)) '.' BASE64URL-ENCODE(JWS Payload)).
- Compute the signature over the message constructed in the previous step, following the signature algorithm defined in the corresponding header element: ***alg***. This element can be either inside the JWS protected header or the JWS unprotected header. The message is signed using the private key corresponding to the public key advertised in the header.
- Compute the base64url encoded value of the JWS signature produced in the previous step, which will produce the value of the ***signature*** element inside the ***signatures*** JSON array of the serialized JWS token.
- Once all the signatures are computed, the ***signatures*** JSON array can be constructed and will complete the JWS JSON serialization.

JWE (JSON Web Encryption)

The JWE (JSON Web Encryption) specification standardizes the way to represent an encrypted content in a JSON-based data structure. It defines two serialized forms to represent the encrypted payload: the ***JWE compact serialization*** and ***JWE JSON serialization***. Both of these two serialization techniques are discussed in detail, in the sections to follow. Like in JWS, **the message to be encrypted using JWE standard needs not to be a JSON payload, it can be any content**.

JWE Compact Serialization

With the JWE compact serialization, a JWE token is built with five key components, each separated by a period (.): ***JOSE header***, ***JWE Encrypted Key***, ***JWE initialization vector***, ***JWE Additional Authentication Data (AAD)***, ***JWE Ciphertext*** and ***JWE Authentication Tag***.



The structure of a JWE token formed by JWE compact serialization.

The JOSE header is the very first element of the JWE token produced under compact serialization. The structure of the JOSE header is the same, as we discussed under JWS other than couple of exceptions. The JWE specification introduces two new elements (***enc*** and ***zip***), which are included in the JOSE header of the JWE token, in addition to what's defined by the JSON Web Signature (JWS) specification.

To understand **JWE Encrypted Key** section of the JWE, we first need to understand how a JSON payload gets encrypted. The ***enc*** element of the JOSE header defines the ***content encryption algorithm*** and it should be a symmetric ***Authenticated Encryption with Associated Data (AEAD)*** algorithm. The ***alg*** element of the JOSE header defines the encryption algorithm to encrypt the ***Content Encryption Key (CEK)***. This algorithm can also be defined as the key wrapping algorithm, as it wraps the ***CEK***.

> ***Authenticated Encryption with Associated Data (AEAD)*** *is a block cipher mode of operation which simultaneously provides confidentiality, integrity, and authenticity assurances on the data; decryption is combined in single step with integrity verification.*

Let’s look at the following JOSE header. For content encryption, it uses ***A256GCM*** algorithm; and for key wrapping, ***RSA-OAEP***:

> *{“alg”:”RSA-OAEP”,”enc”:”A256GCM”}*

A256GCM is defined in the JWA specification. It uses the ***Advanced Encryption Standard (AES) in Galois/Counter Mode (GCM)*** algorithm with a 256-bit long key, and it’s a symmetric key algorithm used for ***AEAD***. Symmetric keys are mostly used for content encryption and it is much faster than asymmetric-key encryption. At the same time, asymmetric-key encryption can’t be used to encrypt large messages.

RSA-OAEP is too defined in the JWA specification. During the encryption process, the token issuer generates a random key, which is 256 bits in size and encrypts the message using that key following the ***AES GCM*** algorithm. Next, the key used to encrypt the message is encrypted using ***RSA-OAEP***, which is an asymmetric encryption scheme. The ***RSA-OAEP*** encryption scheme uses ***RSA*** algorithm with the ***Optimal Asymmetric Encryption Padding (OAEP)*** method. Finally the encrypted symmetric key is placed in the **JWE Encrypted Header** section of the JWE.

Some encryption algorithms, which are used for content encryption require an initialization vector, during the encryption process. Initialization vector is a randomly generated number, which is used along with a secret key to encrypt data. This will add randomness to the encrypted data, which will prevent repetition even the same data gets encrypted using the same secret key again and again. To decrypt the message at the token recipient end, it has to know the initialization vector, hence it is included in the JWE token, under the **JWE Initialization Vector** element. If the content encryption algorithm does not require an initialization vector, then the value of this element should be kept empty.

The fourth element of the JWE token is the base64url-encoded value of the JWE ciphertext. The JWE ciphertext is computed by encrypting the plaintext JSON payload using the ***Content Encryption Key (CEK)***, the ***JWE initialization vector*** and the ***Additional Authentication Data (AAD)*** value, with the encryption algorithm defined by the header element ***enc***. The algorithm defined by the **enc** header element should be a symmetric ***Authenticated Encryption with Associated Data (AEAD)*** algorithm. The ***AEAD*** algorithm, which is used to encrypt the plaintext payload, also allows specifying ***Additional Authenticated Data (AAD)***.

The base64url-encoded value of the **JWE Authenticated Tag** is the final element of the JWE token. As discussed before the value of the authentication tag is produced during the ***AEAD*** encryption process, along with the ciphertext. The authentication tag ensures the integrity of the ciphertext and the ***Additional Authenticated Data (AAD)***.

JWE Compact Serialization—Signing Process

Following lists out the encryption process of a JWE under the compact serialization.

- Figure out the key management mode by the algorithm used to determine the ***Content Encryption Key (CEK)*** value. This algorithm is defined by the ***alg*** element in the JOSE header. There is only one ***alg*** element per JWE token.
- Compute the ***CEK*** and calculate the ***JWE Encrypted Key*** based on the key management mode, picked in the previous. The ***CEK*** is later used to encrypt the JSON payload. There is only one ***JWE Encrypted Key*** element in the JWE token.
- Compute the base64url-encoded value of the ***JWE Encrypted Key***, which is produced in the previous step. This is the 2nd element of the JWE token.
- Generate a random value for the ***JWE Initialization Vector***. Irrespective of the serialization technique, the JWE token will carry the value of the base64url-encoded value of the ***JWE Initialization Vector***. This is the 3rd element of the JWT token.
- If token compression is needed, the JSON payload in plaintext must be compressed following the compression algorithm defined under the ***zip*** header element.
- Construct the JSON representation of the JOSE header and find the base64url-encoded value of the JOSE header with UTF8 encoding. This is the 1st element of the JWE token.
- To encrypt the JSON payload, we need the ***CEK*** (which we already have), the ***JWE Initialization Vector*** (which we already have), and the ***Additional Authenticated Data (AAD)***. Compute ASCII value of the encoded JOSE header from the previous step and use it as the ***AAD***.

- Encrypt the compressed JSON payload (from the previous step) using the ***CEK***, the ***JWE Initialization Vector*** and the ***Additional Authenticated Data (AAD)***, following the content encryption algorithm defined by the header ***enc*** header element.
- The algorithm defined by the ***enc*** header element is a ***AEAD*** algorithm and after the encryption process, it produce the ciphertext and the ***Authentication Tag***.
- Compute the base64url-encoded value of the ciphertext, which is produced by the step one before the previous. This is the 4th element of the JWE token.
- Compute the base64url-encoded value of the Authentication Tag, which is produced by the step one before the previous. This is the 5th element of the JWE token.
- Now we have all the elements to build the JWE token in the following manner. The line breaks are introduced only for clarity.

> *BASE64URL-ENCODE(UTF8(JWE Protected Header)) '.'*

> *BASE64URL-ENCODE(JWE Encrypted Key) '.'*

> *BASE64URL-ENCODE(JWE Initialization Vector) '.'*

> *BASE64URL-ENCODE(JWE Ciphertext) '.'*

> *BASE64URL-ENCODE(JWE Authentication Tag)*

JWE JSON Serialization

Unlike the JWE compact serialization, the JWE JSON serialization can produce encrypted data targeting at multiple recipients over the same JSON payload. The ultimate serialized form under JWE JSON serialization represents an encrypted payload in a JSON object. This JSON object includes six top-level elements: ***protected***, ***unprotected***, ***recipients***, ***iv***, ***ciphertext*** and ***tag***. Following is an example of a JWE token, which is serialized under JWE JSON serialization.

```
{ ⊟
   "protected":"eyJlbmMiOiJBMTI4Q0JDLUhTMjU2In0",
   "unprotected":{ ⊟
      "jku":"https://server.example.com/keys.jwks"
   },
   "recipients":[ ⊟
      { ⊟
         "header":{ ⊟
            "alg":"RSA1_5",
            "kid":"2011-04-29"
         },
         "encrypted_key":"UGhIOguC7IuEvf_NPVaXsGMoLOmwvc1GyqlI9XShH59_i8J0PH5ZZyNfGy2xGd"
      },
      { ⊟
         "header":{ ⊟
            "alg":"A128KW",
            "kid":"7"
         },
         "encrypted_key":"6KB707dM9YTIgHtLvtgWQ8mKwboJW3of9locizkDTHzBC2IlrT1oOQ"
      }
   ],
   "iv":"AxY8DCtDaGlsbGljb3RoZQ",
   "ciphertext":"KDlTtXchhZTGufMYmOYGS4HffxPSUrfmqCHXaI9wOGY",
   "tag":"Mz-VPPyU4RlcuYv1IwIvzw"
}
```

The JWE Protected Header

The JWE protected header is a JSON object that includes the header elements that has to be integrity protected by the authenticated encryption operation (***AEAD)***. The elements inside the JWE protected header are applicable to all the recipients of the JWE token. The ***protected*** element in the serialized JSON form represents the base64url-encoded value of the JWE protected header. There can be only one ***protected*** element in a JWE token at the root level and any header elements that we discussed before under the JOSE header can also be used under the JWE protected header.

JWE Shared Unprotected Header

The JWE shared unprotected header is a JSON object that includes the header elements that are not integrity protected. The elements inside the JWE shared unprotected header are applicable to all the recipients of the JWE token. The ***unprotected*** element in the serialized JSON form represents the JWE shared unprotected header. There can be only one ***unprotected*** element in a JWE token, at the root level and any header element that we discussed before under the JOSE header can also be used under the JWE shared unprotected header.

JWE Per-Recipient Unprotected Header

The JWE per-recipient unprotected header is a JSON object that includes the header elements that are not integrity protected. The elements inside the JWE per-recipient unprotected header are applicable only to a particular recipient of the JWE token. In the JWE token, these header elements are grouped under the element, ***recipients***. The ***recipients*** element represents an array of recipients of the JWE token. Each member consists of a ***header*** element and an ***encrypted_key*** element.

- ***header***: The ***header*** element, which is inside each entry of the ***recipients*** JSON array, represents the value of the corresponding JWE header elements that aren't protected by authenticated encryption for each recipient.
- ***encrytedkey***: The ***encrytedkey*** element represents the base64url-encoded value of the encrypted key. This is the key used to encrypt the message payload. The key can be encrypted in different ways for each recipient.

Any header element that we discussed before under the JOSE header can also be used under the JWE per-recipient unprotected header.

JWE Initialization Vector

This carries the same meaning as explained under JWE compact serialization, previously. The ***iv*** element in the JWE token represents the value of the initialization vector used for encryption.

JWE Ciphertext

This carries the same meaning as explained under JWE compact serialization, previously. The ***ciphertext*** element in the JWE token carries the base64url-encoded value of the JWE ciphertext

JWE Authentication Tag

This carries the same meaning as explained under JWE compact serialization, previously. The ***tag*** element in the JWE token carries the base64url-encoded value of the JWE authenticated tag, which is an outcome of the encryption process using an ***AEAD*** algorithm.

JWE JSON Serialization — Signing Process

Following lists out the encryption process of a JWE under the JSON serialization.

- Figure out the key management mode by the algorithm used to determine the ***Content Encryption Key (CEK)*** value. This algorithm is defined by the ***alg*** element in the JOSE header. Under JWE JSON serialization, the JOSE header is built by the union of all the elements defined under the ***JWE Protected Header, JWE***

Shared Unprotected Header and ***Per-Recipient Unprotected Header***. Once included in the ***Per-Recipient Unprotected Header*** the ***alg*** element can be defined per recipient.

- Compute the ***CEK*** and calculate the ***JWE Encrypted Key*** based on the key management mode, picked in the previous step. The ***CEK*** is later used to encrypt the JSON payload.
- Compute the base64url-encoded value of the ***JWE Encrypted Key***, which is produced in the previous step. Once again this is computed per recipient and the resultant value is included in *the* ***Per-Recipient Unprotected Header*** element*,* ***encrytedkey***.
- Perform all three previous steps for each recipient of the JWE token. Each iteration will produce an element in the ***recipients*** JSON array of the JWE token.
- Generate a random value for the ***JWE Initialization Vector***. Irrespective of the serialization technique, the JWE token will carry the value of the base64url-encoded value of the ***JWE Initialization Vector***.
- If token compression is needed, the JSON payload in plaintext, must be compressed following the compression algorithm defined under the ***zip*** header element. The value of the ***zip*** header element can be defined either under the ***JWE Protected Header*** or ***JWE Shared Unprotected Header***.
- Construct the JSON representation of the ***JWE Protected Header, JWE Shared Unprotected Header*** and ***Per-Recipient Unprotected Headers***.
- Compute the base64url-encoded value of the ***JWE Protected Header*** with UTF8 encoding. This value is represented by the ***protected*** element in the serialized JWE token. The ***JWE Protected Header*** is optional and if present there can be only one header. If no JWE header is present, then the value of the ***protected*** element will be empty.
- Generate a value for the ***Additional Authenticated Data (AAD)*** and compute the base64url-encoded value of it. This is an optional step and if it's done, then the base64url-encoded ***AAD*** value will be used as an input element to encrypt the JSON payload, as explained in the next step.
- To encrypt the JSON payload, we need the ***CEK*** (which we already have), the ***JWE Initialization Vector*** (which we already have),

and the ***Additional Authenticated Data (AAD)***. Compute ASCII value of the encoded ***JWE Protected Header*** (step one before the previous) and use it as the AAD. In case the previous step is done and then the value of AAD is computed as ASCII(encoded *JWE Protected Header* '.' BASE64URL-ENCODE(AAD)).

- Encrypt the compressed JSON payload (from step-6) using the ***CEK***, the ***JWE Initialization Vector*** and the ***Additional Authenticated Data*** (AAD from the previous step), following the content encryption algorithm defined by the header ***enc*** header element.
- The algorithm defined by the ***enc*** header element is a ***AEAD*** algorithm and after the encryption process, it produces the ***ciphertext*** and the ***Authentication Tag***.
- Compute the base64url-encoded value of the ***ciphertext***, which is produced in the previous step.
- Compute the base64url-encoded value of the ***Authentication Tag***, which is produced in the step one before the previous.
- Now we have all the elements to build the JWE token under JSON serialization.

JWT Bindings

A binding for a given token defines how to transport the token from one place to another. For example, SAML HTTP redirect binding defines how to transport a SAML request/response over HTTP redirect. The SAML SOAP binding defines how to transport a SAML request/response wrapped in a SOAP enveloper.

JWT does not have a standard binding. But in most of the cases, the JWT is transported over HTTP under the Authorization Bearer header (like in OAuth 2.0).

> *Authorization: Bearer <jwt-token>*

Everything we discussed so far assumed JWT is a bearer token. A bearer token means, who ever owns the token can use it without proving the ownership of the token. Its like cash. If you steal 100 bucks from someone, you do not need to prove how you got it when you want to spend it. If someone steals a bearer token he can just use it as the legitimate owner of it. Whenever you use a bearer token—or transport it from one place to another, it has to be done over a secured medium; user TLS.

JWT with Proof of Possession

The RFC 7800 describes how a JWT can declare that the presenter of the JWT possesses a particular proof-of-possession (PoP) key and how the recipient can cryptographically confirm proof of possession of the key by the presenter. Proof of possession of a key is also sometimes described as the presenter being a holder-of-key. This is analogous to a credit card protected with the owner's signature. Even if someone steals a credit card he cannot use it without proving the ownership. The ownership is proved by the signature. The JWTs issued under holder-of-key confirmation is out-0f-the-scope of this post and possibly we can discuss this in detail in a later post.

Summary

- JWT is used to transport user identity/entitlements between interested parties in a secured manner.
- JWS and JWE are instances of the JWT—when used compact serialization.
- JWS and JWE can be serialized using either the compact serialization or JSON serialization.
- JWT does not define a specific binding, but in practice JWT tokens are transported over HTTPS under the Authorization Bearer header, just as in OAuth 2.0.

References

- JWT : https://tools.ietf.org/html/rfc7519
- JWS: https://tools.ietf.org/html/rfc7516
- JWS: https://tools.ietf.org/html/rfc7515
- Proof-of-Possession Key Semantics for JSON Web Tokens (JWTs) : https://tools.ietf.org/html/rfc7800

Prabath Siriwardena

Visit Amazon.com's Prabath Siriwardena Page and shop for all Prabath Siriwardena books and other…

www.amazon.com