jose Documentation

Release 0.1

Demian Brecht

December 09, 2016

Contents

Contents

CHAPTER 1

Overview

JOSE [1] is a framework intended to provide a method to securely transfer claims (such as authorization information) between parties. The JOSE framework provides a collection of specifications to serve this purpose. A JSON Web Token (JWT) [2] contains claims that can be used to allow a system to apply access control to resources it owns. One potential use case of the JWT is as the means of authentication and authorization for a system that exposes resources through an OAuth 2.0 model [5].

Claims are a set of key/value pairs that provide a target system with sufficient information about the given client to apply the appropriate level of access control to resources under its ownership. Claim names are split into three classes: Registered (IANA), Public and Private. Further details about claims can be found in section 4 of the JWT specification.

JWTs can be represented as either JSON Web Signature (JWS) [3] or a JSON Web Encryption (JWE) [4] objects. Claims within a JWS can be read as they are simply base64-encoded (but carry with them a signature for authentication). Claims in a JWE on the other hand, are encrypted and as such, are entirely opaque to clients using them as their means of authentication and authorization.

[1] JOSE: JSON Object Signing and Encryption
https://datatracker.ietf.org/wg/jose/charter/
[2] JWT: JSON Web Tokens
https://tools.ietf.org/html/draft-ietf-oauth-json-web-token
[5] JWT Authorization Grants
http://tools.ietf.org/html/draft-ietf-oauth-jwt-bearer
[3] JWS: JSON Web Signing
http://tools.ietf.org/html/draft-ietf-jose-json-web-signature
[4] JWE: JSON Web Encryption
http://tools.ietf.org/html/draft-ietf-jose-json-web-encryption

CHAPTER 2

JWK

A JSON Web Key (JWK) [6] is a JSON data structure that represents a cryptographic key. Using a JWK rather than one or more parameters allows for a generalized key as input that can be applied to a number of different algorithms that may expect a different number of inputs. All JWE and JWS operations expect a JWK rather than inflexible function parameters.

2.1 JWK format

```
jwk = {'k': <password>}
```

Currently, the only key/value pair that's required throughout the JWE and JWS flows is *'k'*, indicating the key, or password.

Note: The password must match algorithm requirements (i.e. a key used with an RSA algorithm must be at least 2048 bytes and be a valid private or public key, depending on the cryptographic operation). Other fields may be required in future releases.

[6] JWK: JSON Web Keys
http://tools.ietf.org/html/draft-ietf-jose-json-web-key

CHAPTER 3

JWS

3.1 Definition

A deserialized JWS is represented as a *namedtuple* with the following definition:

3.2 API

3.3 Example

```
import jose

claims = {
    'iss': 'http://www.example.com',
    'exp': int(time()) + 3600,
    'sub': 42,
}

jwk = {'k': 'password'}

jws = jose.sign(claims, jwk, alg='HS256')
# JWS(header='eyJhbGciOiAiSFMyNTYifQ',
# payload='eyJpc3MiOiAiaHR0cDovL3d3dy5leGFtcGxlLmNvbSIsICJzdWIiOiA0MiwgImV4cCI6IDEzOTU2NzQ0Mjd9',
# signature='WYApAiwiKd-eDClA1fg7XFrnfHzUTgrmdRQY4M19Vr8')

# issue the compact serialized version to the clients. this is what will be
# transported along with requests to target systems.

jwt = jose.serialize_compact(jws)
# 'eyJhbGciOiAiSFMyNTYifQ.eyJpc3MiOiAiaHR0cDovL3d3dy5leGFtcGxlLmNvbSIsICJzdWIiOiA0MiwgImV4cCI6IDEzOTU

jose.verify(jose.deserialize_compact(jwt), jwk)
# JWT(header={u'alg': u'HS256'}, claims={u'iss': u'http://www.example.com', u'sub': 42, u'exp': 1395
```

3.4 Algorithm support

Symmetric	HS256, HS384, HS512
Asymmetric	RS256, RS384, RS512

CHAPTER 4

JWE

```
import jose
from time import time
from Crypto.PublicKey import RSA

# key for demonstration purposes
key = RSA.generate(2048)

claims = {
    'iss': 'http://www.example.com',
    'exp': int(time()) + 3600,
    'sub': 42,
}

# encrypt claims using the public key
pub_jwk = {'k': key.publickey().exportKey('PEM')}

jwe = jose.encrypt(claims, pub_jwk)
# JWE(header='eyJhbGciOiAiUlNBLU9BRVAiLCAiZW5jIjogIkExMjhDQkMtSFMyNTYifQ',
# cek='SsLgP2bNKYDYGzHvLYY7rsVEBHSms6_jW-WfglHqD9giJhWwrOwqLZOaoOycsf_EBJCkHq9-vbxRb7WiNdy_C9J0_RnRRI
# iv='Awelp3ryBVpdFhRckQ-KKw',
# ciphertext='1MyZ-3nky1EFO4UgTB-9C2EHpYh1Z-ij0RbiuuMez70nIH7uqL9hlhskutO0oPjqdpmNc9glSmO9pheMH2DVag
# tag='Xcccк85XZMvG-fAJ6oDnAw')

# issue the compact serialized version to the clients. this is what will be
# transported along with requests to target systems.

jwt = jose.serialize_compact(jwe)
# 'eyJhbGciOiAiUlNBLU9BRVAiLCAiZW5jIjogIkExMjhDQkMtSFMyNTYifQ.SsLgP2bNKYDYGzHvLYY7rsVEBHSms6_jW-Wfgl

# decrypt on the other end using the private key
priv_jwk = {'k': key.exportKey('PEM')}

jwt = jose.decrypt(jose.deserialize_compact(jwt), priv_jwk)
# JWT(header={u'alg': u'RSA-OAEP', u'enc': u'A128CBC-HS256'},
# claims={u'iss': u'http://www.example.com', u'sub': 42, u'exp': 1395606273})
```

4.1 Algorithm support

Note: There are two different encryption algorithms employed to fully encrypt a JWE: Encryption of the Content Encryption Key (CEK) and encryption of the JWT claims. The encryption algorithm used to encrypt the CEK is set through the *alg* parameter of `encrypt()` and the claims encryption is defined by the *enc* parameter.

4.1.1 CEK Encryption (*alg*)

Symmetric	[None]
Asymmetric	RSA-OAEP

4.1.2 Claims Encryption (*enc*)

Symmetric	A128CBC-HS256, A192CBC-HS256, A256CBC-HS512
Asymmetric	[N/A]

CHAPTER 5

Serialization

CHAPTER 6

JWT

A `JWT` is a *namedtuple* result produced by either decrypting or verifying a `JWE` or a `JWS`.

CHAPTER 7

Errors

CHAPTER 8

References