RFC 7515 : JSON Web Signature (JWS)

Stéphane Bortzmeyer
<stephane+blog@bortzmeyer.org>

Première rédaction de cet article le 20 mai 2015

Date de publication du RFC : Mai 2015

http://www.bortzmeyer.org/7515.html

Dans la série des RFC JOSE <http://www.bortzmeyer.org/jose.html>, qui décrivent les possibilités de protéger des textes JSON avec la cryptographie, ce RFC 7515 [1] normalise les signatures, qui permettent de garantir l'authenticité et l'intégrité des textes JSON.

Le format JSON, normalisé dans le RFC 8259, ne bénéficiait pas jusqu'alors d'un moyen standard pour permettre la vérification de son authenticité. Certes, on pouvait transport le JSON via un transport sécurisé comme HTTPS mais cela ne fournissait qu'une protection du canal, pas un contrôle d'intégrité de bout en bout.

Le principe de ce RFC est le suivant (section 3) : un texte JSON protégé par une signature, un JWS ("*JSON Web Signature*"), est composé de trois parties : l'**en-tête JOSE** ("*JavaScript Object Signing and Encryption*"), qui indique les paramètres cryptographiques, la **charge utile JWS** (le message d'origine), et la **signature JWS**. L'en-tête JOSE se sépare en un en-tête protégé (par la signature) et un en-tête non protégé. Le document rassemblant ces trois parties est ensuite **sérialisé** en une suite d'octets. Il y a deux sérialisations possibles. La sérialisation compacte est simplement la concaténation de l'encodage en Base64 de l'en-tête protégé, d'un point, de l'encodage en Base64 de la charge utile, un autre point et enfin l'encodage en Base64 de la signature. Un JWS ainsi sérialisé peut être utilisé dans un URL. Voici un exemple :

```
eyJ0eXAiOiJKV1QiLA0KICJhbGciOiJIUzI1NiJ9.eyJpc3MiOiJqb2UiLA0KICJleHAiOjEzMDA4MTkzODAsDQogImh0dHA6Ly9leGFtcGxlLmN
```

1. Pour voir le RFC de numéro NNN, https://www.ietf.org/rfc/rfcNNN.txt, par exemple https://www.ietf.org/rfc/rfc7515.txt

Pas très lisible, évidemment. La deuxième sérialisation possible est dite sérialisation JSON car son résultat est un texte JSON. C'est un objet JSON comportant quatre membres (pas forcément obligatoires), `payload`, la charge utile, `protected`, l'en-tête JOSE protégé, `header`, l'en-tête JOSE non protégé, et `signature`, la signature. Les trois derniers peuvent se retrouver regroupés dans un sous-objet nommé `signatures`. En voici un exemple :

```
{
      "payload":
       "eyJpc3MiOiJqb2UiLA0KICJleHAiOjEzMDA4MTkzODAsDQogImh0dHA6Ly9leGFtcGxlLmNvbS9pc19yb290Ijp0cnVlfQ",
      "signatures":[
       {"protected":"eyJhbGciOiJSUzI1NiJ9",
        "header":
         {"kid":"2010-12-29"},
        "signature":
         "cC4hiUPoj9Eetdgtv3hF80EGrhuB__dzERat0XF9g2VtQgr9PJbu3XOiZj5RZmh7AAuHIm4Bh-0Qc_lF5YKt_O8W2Fp5jujGbo
       {"protected":"eyJhbGciOiJFUzI1NiJ9",
        "header":
         {"kid":"e9bc097a-ce51-4036-9562-d2ade882db0d"},
        "signature":
         "DtEhU3ljbEg8L38VWAfUAqOyKAM6-Xx-F4GawxaepmXFCgfTjDxw5djxLa8ISlSApmWQxfKTUJqPP3-Kg6NU1Q"}]
}
```

Si vous n'avez pas de décodeur Base64 sous la main, notez que la charge utile (le texte JSON qu'on veut protéger) était :

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

Après cette vision générale, les détails. D'abord, l'en-tête JOSE (section 4). On a vu qu'il indiquait les paramètres algorithmiques de la signature. Les plus importants de ces paramètres :

— `alg` indique l'algorithme cryptographique utilisé. Les valeurs possibles dans le registre IANA <https://www.iana.org/assignments/jose/jose.xhtml#web-signature-encryption-algori défini par le RFC 7518. Par exemple {"alg":"ES256"} identifiera l'algorithme ECDSA avec la courbe P-256 et la condensation en SHA-256.
— `jwk` est la clé de signature, formatée selon le RFC 7517.
— `jku` est un URL au bout duquel on trouvera un ensemble de clés, formatées également selon le RFC 7517. La clé de signature fait partie de cet ensemble. Évidemment, il faut utiliser un transport sécurisé (comme HTTPS, avec validation de l'identité du serveur, cf. RFC 6125) pour récupérer ces clés (section 8, sur les exigences TLS).
— `kid`, que vous avez vu dans l'exemple plus haut, est un identificateur de la clé de signature (cf. section 6).
— `x5c` est un certificat X.509 (RFC 5280) correspondant à la clé de signature.
— `x5u` est un URL pointant vers un certificat X.509 correspondant à la clé de signature.

Les étapes pour créer un JWS sont décrites en section 5 :

— Encoder la charge utile en Base64 (RFC 4648). Notez qu'il n'existe pas de mécanisme de canonicalisation standard d'un texte JSON et qu'on n'applique donc pas une telle canonicalisation.
— Fabriquer l'en-tête JOSE avec les paramètres cryptographiques qu'on va utiliser, puis l'encoder en Base64.
— Signer (les détails dépendent évidemment de l'algorithme de signature utilisé), et encoder la signature en Base64.
— Sérialiser (en JSON ou en compact).

Pour valider une signature, on fait l'inverse. Si une étape échoue, la signature est rejetée.

— Analyser la forme sérialisée pour en extraire les différents composants (en-tête, charge utile, signature).

— Décoder (depuis le Base64) l'en-tête protégé et vérifier qu'on obtient bien du JSON correct.
— Vérifier qu'on comprend tous les paramètres, par exemple l'algorithme de signature utilisé.
— Décoder (depuis le Base64) la signature et la charge utile.
— Valider la (ou les) signature(s).

S'il y a plusieurs signatures, ce que permet JWS, doivent-elles être toutes valides ou bien une seule suffit-elle ? Ce n'est pas défini par la norme JOSE : chaque application utilisant JOSE peut décider comme elle veut. Dans tous les cas, pour que le document soit considéré comme correctement signé, il faut qu'au moins une signature soit valide. Une autre décision est du ressort de l'application : quels sont les algorithmes de signature acceptables. Par exemple, une application peut décider de rejeter une signature car l'algorithme, trop faible cryptographiquement parlant, n'est pas accepté.

Notez que pas mal d'étapes de JOSE nécessitent de comparer des chaînes de caractères. Cela soulève immédiatement les problèmes de canonicalisation. Les règles JSON de comparaison pour les noms des membres d'un objet sont dans la section 8.3 du RFC 8259. Pour les valeurs, c'est plus compliqué et cela dépend de l'application. Ainsi, si une application a décidé que le membre `kid` (identificateur de la clé) contient un nom de domaine, elle peut faire des comparaisons insensibles à la casse.

La section 7 de notre RFC couvre les sérialisations. Comme on l'a vu plus haut, il y a deux sérialisations, la compacte et celle en JSON. Chaque application de JOSE va décider quelles sérialisations sont acceptables, la compacte, la JSON ou les deux. La sérialisation en JSON a une forme générale, où les signatures sont un tableau (éventuellement de taille 1) et une légère variante, la sérialisation en JSON aplati, où une seule signature est possible. Voici les deux cas, d'abord la forme générale, avec deux signatures :

```
{
      "payload":"...",
      "signatures":[
       {"protected":"...",
        "signature":"..."},
       ...
       {"protected":"...",
        "signature":"..."}]
     }
```

Puis la forme aplatie :

```
{
      "payload":"...",
      "protected":"...",
      "signature":"..."
     }
```

La sérialisation compacte, lorsqu'elle est envoyée via l'Internet, est étiquetée avec le type MIME `application/jose`. La sérialisation en JSON est `application/jose+json`. Ces deux types sont utilisables pour les JWS de ce RFC ou pour les JWE du RFC 7516.

La section 10 de notre RFC revient sur les exigences de sécurité, qui accompagnent tout protocole cryptographique. La plupart ne sont pas spécifiques à JOSE. Par exemple, si on laisse trainer sa clé privée n'importe où, des tas de gens pourront faire des fausses signatures. Autre cas bien connu, le générateur aléatoire utilisé pour générer les clés (et autres valeurs qui doivent être imprévisibles) doit être fiable (des tas de failles de sécurité ont été créées par des générateurs insuffisants). Mais certains problèmes sont un peu plus subtils. Par exemple, supposons qu'on valide la signature et que, cryptographiquement, tout soit au point. Mais qu'est-ce que ça nous prouve ? Ça nous prouve que le document a bien été signé par

le titulaire de la clé privée utilisée. Mais, en général, on veut plus que cela. On veut que le document soit bien signé par Mme A ou par la société B. Il y a donc une étape entre la clé et l'entité à qui on fait confiance ("*Key Origin Authentication*") qui n'est pas triviale.

Autre point qui n'est pas forcément maitrisé de tous, la différence entre signature et MAC. La section 10.5 l'explique bien : la clé utilisée pour MAC est entre les mains de plusieurs entités et l'authentification garantie par MAC est donc plus faible.

Il peut y avoir aussi des failles de sécurité dans JSON et pas dans JOSE, notamment en raison du flou de cette norme, et du laxisme de certaines mises en œuvre qui acceptent du JSON mal formé. Ainsi, le RFC 8259 n'impose pas que les membres d'un objet soient uniques. Un objet JSON :

```
{
  "voter": "Smith",
  "vote": "YES",
  "vote": "NO"
}
```

est légal (quoique très déconseillé). Si on sait qu'un utilisateur de cet objet a un analyseur qui ne garde que le dernier membre dupliqué, et qu'un autre a un analyseur qui ne garde que le premier membre dupliqué, on peut signer cet objet et faire croire au premier utilisateur que Smith a voté non, et au second que le même Smith a voté oui. Pour éviter cela, la norme JOSE durcit le RFC 8259 en interdisant les membres dupliqués (une meilleure solution aurait été d'imposer l'utilisation du sous-ensemble JSON du RFC 7493, je trouve, mais l'IETF a choisi une autre voie <http://www.ietf.org/mail-archive/web/secdir/current/msg05105.html>). Une conséquence de ce durcissement est qu'on ne peut pas signer tous les objets JSON. Les cas pathologiques sont exclus. (Ce point a été l'un des plus « chauds » dans la mise au point de JOSE à l'IETF.)

Plein d'examples de signatures se trouvent dans l'annexe A de notre RFC (mais qu'on avait aussi dans le RFC 7520).