RFC 7520 : Examples of Protecting Content using JavaScript Object Signing and Encryption (JOSE)

Stéphane Bortzmeyer
<stephane+blog@bortzmeyer.org>

Première rédaction de cet article le 20 mai 2015

Date de publication du RFC : Mai 2015

https://www.bortzmeyer.org/7520.html

Ce RFC fait partie de l'l'ensemble des normes <https://www.bortzmeyer.org/jose.html> sur le système JOSE ("*JavaScript Object Signing and Encryption*", opérations cryptographiques sur les textes JSON). Il présente des exemples d'utilisation de JOSE, permettant aux développeurs de voir s'ils ont bien compris les normes présentées dans les RFC 7515 [1] et RFC 7516.

Composé essentiellement de textes JSON et de binaire, ce RFC est long : 117 pages. Je n'en décris ici qu'une toute petite partie. Il vaux mieux lire avant les autres RFC sur JOSE <https://www.bortzmeyer.org/jose.html> pour comprendre les exemples. JOSE est très riche et il existe plein d'options. Il n'est donc pas possible, malgré l'épaisseur de ce RFC, de couvrir tous les cas. Le but est d'avoir au moins un échantillon représentatif des cas les plus courants. JOSE fonctionne en deux temps : calcul des résultats cryptographiques puis **sérialisation**, qui est la représentation sous forme texte, de ces résultats. JOSE a trois sérialisations possibles : compacte (le résultat est une série de caractères, pas un texte JSON), générale (le résultat est lui-même du JSON) et aplatie (également du JSON). Si vous voulez vérifier les exemples vous-même, ils sont tous disponibles sur GitHub <https://github.com/ietf-jose/cookbook>.

Normalement, le contenu du RFC suffit à reproduire l'exemple exactement : en le donnant à une mise en œuvre de JOSE, on doit trouver le même résultat. La principale exception concerne le cas des algorithmes qui utilisent en entrée des données aléatoires (comme DSA). Dans ce cas, il n'est pas possible de retrouver exactement les mêmes valeurs. En outre, pour la présentation dans le RFC, des espaces ont parfois été ajoutés, là où il n'affectait pas la sémantique des opérations cryptographiques.

Commençons par les premiers exemples (section 3), la représentation de clés en JOSE (RFC 7517). D'abord, une clé publique, sur une courbe elliptique, la P-521 :

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

```
{
     "kty": "EC",
     "kid": "bilbo.baggins@hobbiton.example",
     "use": "sig",
     "crv": "P-521",
     "x": "AHKZLLOsCOzz5cY97ewNUajB957y-C-U88c3v13nmGZx6sYl_oJXu9A5RkTKqjqvjyekWF-7ytDyRXYgCF5cj0Kt",
     "y": "AdymlHvOiLxXkEhayXQnNCvDX4h9htZaCJN34kfmC6pV5OhQHiraVySsUdaQkAgDPrwQrJmbnX9cwlGfP-HqHZR1"
   }
```

Ici, `x` et `y` sont des paramètres spécifiques des courbes elliptiques, encodés en Base64 (JSON n'a pas de moyen plus pratique de stocker du binaire). Si c'était une clé privée, on aurait également un paramètre `d`. `kid` est l'identificateur de la clé. Plus banale que les courbes elliptiques, une clé publique RSA :

```
   {
     "kty": "RSA",
     "kid": "bilbo.baggins@hobbiton.example",
     "use": "sig",
     "n": "n4EPtAOCc9AlkeQHPzHStgAbgs7bTZLwUBZdR8_KuKPEHLd4rHVTeT-O-XV2jRojdNhxJWTDvNd7nqQ0VEiZQHz_AJmSCpMaJ
     "e": "AQAB"
   }
```

Ici, `n` et `e` sont des paramètres spécifiques de RSA (le module et l'exposant). Une clé privée RSA nécessiterait davantage d'informations :

```
   {
     "kty": "RSA",
     "kid": "bilbo.baggins@hobbiton.example",
     "use": "sig",
     "n": "n4EPtAOCc9AlkeQHPzHStgAbgs7bTZLwUBZdR8_KuKPEHLd4rHVTeT-O-XV2jRojdNhxJWTDvNd7nqQ0VEiZQHz_AJmSCpMaJ
     "e": "AQAB",
     "d": "bWUC9B-EFRIo8kpGfh0ZuyGPvMNKvYWNtB_ikiH9k20eT-O1q_I78eiZkpXxXQ0UTEs2LsNRS-8uJbvQ-A1irkwMSMkK1J3XT
     "p": "3Slxg_DwTXJcb6095RoXygQCAZ5RnAvZlno1yhHtnUex_fp7AZ_9nRaO7HX_-SFfGQeutao2TDjDAWU4Vupk8rw9JR0AzZ0N2
     "q": "uKE2dh-cTf6ERF4k4e_jy78GfPYUIaUyoSSJuBzp3Cubk3OCqs6grT8bR_cu0Dm1MZwWmtdqDyI95HrUeq3MP15vMMON8lHTe
     "dp": "B8PVvXkvJrj2L-GYQ7v3y9r6Kw5g9SahXBwsWUzp19TVlgI-YV85q1NIb1rxQtD-IsXXR3-TanevuRPRt5OBOdiMGQp8pbt2
     "dq": "CLDmDGduhylc9o7r84rEUVn7pzQ6PF83Y-iBZx5NT-TpnOZKF1pErAMVeKzFEl41DlHHqqBLSM0W1sOFbwTxYWZDm6sI6og5
     "qi": "3PiqvXQN0zwMeE-sBvZgi289XP9XCQF3VWqPzMKnIgQp7_Tugo6-NZBKCQsMf3HaEGBjTVJs_jcK8-TRXvaKe-7ZMaQj8VfB
   }
```

À noter que la seule indication qu'il s'agisse d'une clé privée est la présence des membres supplémentaires `p`, `q`, `dp`, `dq` et `qi`.

Passons maintenant à la cryptographie symétrique. Cette fois, il n'y a qu'une seule clé, stockée dans le membre `k` :

```
{
     "kty": "oct",
     "kid": "018c0ae5-4d9b-471b-bfd6-eef314bc7037",
     "use": "sig",
     "alg": "HS256",
     "k": "hJtXIZ2uSN5kbQfbtTNWbpdmhkV8FJG-Onbc6mxCcYg"
   }
```

Elle sera utilisée pour les signatures : on concatène la valeur de la clé au contenu et on SHA-256 le tout (non, en fait, c'est plus compliqué, voir RFC 2104). Le vérificateur fait la même opération et doit trouver le même condensat SHA-256.

Voici pour les clés. Passons aux signatures, en section 4. Un texte JSON signé est un JWS ("*JSON Web Signature*"). Le texte signé de tous les exemples est un extrait du Seigneur des anneaux, « "*It[Caractère Unicode non montré [2]]s a dangerous business, Frodo, going out your door. You step onto the road, and if you don't keep your feet, there[Caractère Unicode non montré]s no knowing where you might be swept off to.*" ». Attention si vous recalculez les signatures vous-même, le caractère qui ressemble à une apostrophe est en fait le caractère Unicode U+2019, « "*RIGHT SINGLE QUOTATION MARK*" » (et, dans le RFC, il y a en plus remplacement de certains espaces par des sauts de ligne). Si vous voulez être sûr, l'encodage en Base64 (RFC 4648) de cette charge utile est `SXTigJlzIGEgZGFuZ2Vyb3VzIGJ1c2luZXNzLCBGcm9kbywgZ29pbmcgb3V0IHlvdXIg`

Voici d'abord une bête signature RSA. L'en-tête protégé (cf. RFC 7515, sections 2 et 4) est :

```
{
     "alg": "RS256",
     "kid": "bilbo.baggins@hobbiton.example"
}
```

On signe la concaténation du Base64 de cet en-tête et du Base64 de la charge utile. Cela donne `MRjdkly7_-oTPTS3AXP41iQIGKa80A0ZmTuV5MEaHoxnW2e5CZ5NlKtainoFmKZopdHM1O2U4mwzJdQx996ivp83xu qHRD68BNt1uSNCrUCTJDt5aAE6x8wW1Kt9eRo4QPocSadnHXFxnt8Is9UzpERV0ePPQdLuW3IS_- de3xyIrDaLGdjluPxUAhb6L2aXic1U12podGU0KLUQSE_oI-ZnmKJ3F4uOZDnd6QZWJushZ41Axf_- fcIe8u9ipH84ogoree7vjbU5y18kDquDg`. Il reste à **sérialiser** le résultat, c'est-à-dire à le présenter sous une forme concrète. Le RFC 7515 prévoit plusieurs sérialisations possibles. La compacte, utilisable dans les URL, serait `eyJhbGciOiJSUzI1NiIsImtpZCI6ImJpbGJvLmJhZ2dpbnNAaG9iYml0b24uZXhhbXBsZSJ9.SX -oTPTS3AXP41iQIGKa80A0ZmTuV5MEaHoxnW2e5CZ5NlKtainoFmKZopdHM1O2U4mwzJdQx996ivp83xuglII7PNDi qHRD68BNt1uSNCrUCTJDt5aAE6x8wW1Kt9eRo4QPocSadnHXFxnt8Is9UzpERV0ePPQdLuW3IS_- de3xyIrDaLGdjluPxUAhb6L2aXic1U12podGU0KLUQSE_oI-ZnmKJ3F4uOZDnd6QZWJushZ41Axf_- fcIe8u9ipH84ogoree7vjbU5y18kDquDg` (vous reconnaissez la charge utile, la phrase de Tolkien, et la signature, séparées par des points). Avec la sérialisation aplatie, qui donne un texte JSON, ce serait :

```
{
     "payload": "SXTigJlzIGEgZGFuZ2Vyb3VzIGJ1c2luZXNzLCBGcm9kbywgZ29pbmcgb3V0IHlvdXIgZG9vci4gWW91IHN0ZXAgb250byB
     "protected": "eyJhbGciOiJSUzI1NiIsImtpZCI6ImJpbGJvLmJhZ2dpbnNAaG9iYml0b24uZXhhbXBsZSJ9",
     "signature": "MRjdkly7_-oTPTS3AXP41iQIGKa80A0ZmTuV5MEaHoxnW2e5CZ5NlKtainoFmKZopdHM1O2U4mwzJdQx996ivp83xuglI
   }
```

C'est un peu compliqué avec une signature ECDSA. En effet, cet algorithme exige des données imprévisibles (le RFC dit « aléatoires » mais il exagère, cf. RFC 6979) et donc chaque signature donnera un résultat différent. Ne vous étonnez donc pas si vous recalculez et que vous trouvez une valeur différente de celle du RFC. Ici, on réutilise la clé présentée plus haut (sur la courbe P-521), et, en sérialisation générale (qui est très proche de la sérialisation aplatie, elle donne aussi un texte JSON, mais elle a un tableau de signatures et pas une signature unique, voir la section 4.8 pour des exemples) :

```
{
     "payload": "SXTigJlzIGEgZGFuZ2Vyb3VzIGJ1c2luZXNzLCBGcm9kbywgZ29pbmcgb3V0IHlvdXIgZG9vci4gWW91IHN0ZXAgb250byB
     "signatures": [
       {
          "protected": "eyJhbGciOiJFUzUxMiIsImtpZCI6ImJpbGJvLmJhZ2dpbnNAaG9iYml0b24uZXhhbXBsZSJ9",
          "signature": "AE_R_YZCChjn4791jSQCrdPZCNYqHXCTZH0-JZGYNlaAjP2kqaluUIIUnC9qvbu9Plon7KRTzoNEuT4Va2cmL1eJ
       }
     ]
   }
```

2. Car trop difficile à faire afficher par LaTeX

On peut aussi avoir des signatures « détachées », où le texte signé n'apparait pas. Le vérificateur devra récupérer le texte JSON signé, et récupérer la signature détachée, avant de vérifier. Ici, on utilise la clé symétrique montrée plus haut. L'en-tête sera donc :

```
{
     "alg": "HS256",
     "kid": "018c0ae5-4d9b-471b-bfd6-eef314bc7037"
   }
```

Et le résultat, en sérialisation aplatie :

```
{
     "protected": "eyJhbGciOiJIUzI1NiIsImtpZCI6IjAxOGMwYWU1LTRkOWItNDcxYi1iZmQ2LWVlZjMxNGJjNzAzNyJ9",
     "signature": "s0h6KThzkfBBBkLspW1h84VsJZFTsPPqMDA7g1Md7p0"
}
```

Passons maintenant au chiffrement (section 5). Le texte utilisé pour les exemples change. Il est également tiré de « La communauté de l'anneau » et est « *"You can trust us to stick with you through thick and thin[Caractère Unicode non montré]to the bitter end. And you can trust us to keep any secret of yours[Caractère Unicode non montré]closer than you keep it yourself. But you cannot trust us to let you face trouble alone, and go off without a word. We are your friends, Frodo."* » Le tiret est un U+2013, « *"EN DASH"* ».

Premier exemple, avec RSA pour le chiffrement asymétrique de la clé (de la CEK, "*Content Encryption Key*"), AES (en mode CBC) pour le chiffrement symétrique, et SHA-256 lorsqu'il faut condenser. Voici la clé publique :

```
 {
     "kty": "RSA",
     "kid": "frodo.baggins@hobbiton.example",
     "use": "enc",
     "n": "maxhbsmBtdQ3CNrKvprUE6n9lYcregDMLYNeTAWcLj8NnPU9XIYegTHVHQjxKDSHP2l-F5jS7sppG1wgdAqZyhnWvXhYNvcM
     "e": "AQAB"
}
```

L'en-tête protégé est :

```
{
     "alg": "RSA1_5",
     "kid": "frodo.baggins@hobbiton.example",
     "enc": "A128CBC-HS256"
}
```

Et on se retrouve avec, en sérialisation générale :

```
{
     "recipients": [
       {
         "encrypted_key": "laLxI0j-nLH-_BgLOXMozKxmy9gffy2gTdvqzfTihJBuuzxg0V7yk1WClnQePFvG2K-pvSlWc9BRIazDr
       }
     ],
     "protected": "eyJhbGciOiJSU0ExXzUiLCJraWQiOiJmcm9kby5iYWdnaW5zQGhvYmJpdG9uLmV4YW1wbGUiLCJlbmMiOiJBMTI4Q
     "iv": "bbd5sTkYwhAIqfHsx8DayA",
     "ciphertext": "0fys_TY_na7f8dwSfXLiYdHaA2DxUjD67ieF7fcVbIR62JhJvGZ4_FNVSiGc_raa0HnLQ6s1P2sv3Xzl1p1l_o5w
     "tag": "kvKuFBXHe5mQr4lqgobAUg"
}
```

Le contenu chiffré, et donc désormais confidentiel, est dans le membre `ciphertext`. Attention si vous essayez de reproduire cet exemple, le résultat exact dépend du vecteur d'initialisation `iv`. En sérialisation compacte, on aurait :

eyJhbGciOiJSU0ExXzUiLCJraWQiOiJmcm9kby5iYWdnaW5zQGhvYmJpdG9uLmV4YW1wbGUiLCJlbmMiOiJBMTI4Q0JDLUhTMjU2In0.laLxI0j-

On peut aussi chiffrer **et** signer (section 6).

Attention, les exemples dans ce RFC sont juste... des exemples. Ils visent à permettre aux développeurs de comprendre la norme, et de tester leurs programmes (section 7). L'accent est donc mis, autant que possible, sur la reproductibilité et pas sur la sécurité. Ainsi, en violation des bonnes pratiques, tous les exemples utilisent la même clé symétrique de chiffrement (CEK, "*Content Encryption Key*") et le même vecteur d'initialisation. Cette clé et ce vecteur doivent normalement changer à chaque opération de signature ou de chiffrement.

N'oubliez pas, si vous voulez reproduire les exemples de ce RFC, tous les résultats, sous un format lisible par vos programmes, sont dans un dépôt Github <https://github.com/ietf-jose/cookbook>.