RFC 8785 : JSON Canonicalization Scheme (JCS)

Stéphane Bortzmeyer
<stephane+blog@bortzmeyer.org>

Première rédaction de cet article le 29 juin 2020

Date de publication du RFC : Juin 2020

https://www.bortzmeyer.org/8785.html

Des opérations cryptographiques comme la signature sont nettement plus simples lorsqu'il existe une forme **canonique** des données, une représentation normalisée qui fait que toutes les données identiques auront la même représentation. Le format de données JSON n'a pas de forme canonique standard, ce RFC documente une des canonicalisations possibles, JCS ("*JSON Canonicalization Scheme*").

Pourquoi canonicaliser ? Parce que deux documents JSON dont le contenu est identique peuvent avoir des représentations texte différentes. Par exemple :

```
{"foo": 1, "bar": 3}
```

et

```
{
  "foo": 1,
  "bar": 3
}
```

représentent exactement le même objet JSON. Mais la signature de ces deux formes textuelles ne donnera pas le même résultat. Pour les mécanismes de signature de JSON (comme JWS, décrit dans le RFC 7515 [1]), il serait préférable d'avoir un moyen de garantir qu'il existe une **représentation canonique** de l'objet JSON. Ce RFC en propose une. Je vous le dis tout de suite, la forme canonique de l'objet ci-dessus sera :

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

```
{"bar":3,"foo":1}
```

JSON est normalisé dans le RFC 8259. Il n'a pas de canonicalisation standard. Comme c'est une demande fréquente, plusieurs définitions d'une forme canonique de JSON ont été faites (cf. annexe H du RFC), comme "*Canonical JSON*" <http://wiki.laptop.org/go/Canonical_JSON> ou comme "*JSON Canonical Form*" <http://gibson042.github.io/canonicaljson-spec/>. Mais aucune n'est encore **la** référence. (Vous noterez que ce RFC n'est pas une norme.) Cette situation est assez embêtante, alors que XML, lui, a une telle norme (cf. XML Signature <https://www.w3.org/TR/xmldsig-core1/>).

Notez que la solution décrite dans ce RFC se nomme JCS, mais qu'il y a aussi un autre JCS, "*JSON Cleartext Signature*".

Parmi les concepts importants de la proposition de ce RFC, JCS :

— Réutiliser les règles de sérialisation de la norme ECMA, connues sous le nom de « ES6 <https://www.ecma-international.org/ecma-262/6.0/index.html> »,
— Le JSON doit suivre le sous-ensemble I-JSON du RFC 7493, ce qui implique l'absence de duplication des noms de membres dans un objet, l'utilisation de IEEE-754 pour les nombres, etc.

La section 3 du RFC décrit les opérations qui, ensemble, forment la canonicalisation JCS. À partir de données en mémoire (obtenues soit en lisant un fichier JSON, ce que je fais dans les exemples à la fin, soit en utilisant des données du programme), on émet du JSON avec :

— Pas d'espaces entre les éléments lexicaux (`"foo":1` et pas `"foo": 1`),
— Représentation des nombres en suivant les règles de sérialisation « ES6 » citées plus haut (`3000000` au lieu de `3E6`, c'est la section 7.12.2.1 de ES6 <https://www.ecma-international.org/ecma-262/6.0/index.html#sec-tostring>, si vous avez le courage, les puissances de 10 où l'exposant est inférieur à 21 sont développées),
— Membres des objets (rappel : un objet JSON est un dictionnaire) triés ({`"a":true,"b":false`} et non pas {`"b:"false,"a":true`}),
— Chaînes de caractères en UTF-8.

Concernant le tri, André Sintzoff me fait remarquer que les règles de tri de JSON sont parfois complexes. Évidentes pour les caractères ASCII, elles sont plus déroutantes en dehors du BMP.

Plusieurs annexes complètent ce RFC. Ainsi, l'annexe B fournit des exemples de canonicalisation des nombres.

Et pour les travaux pratiques, on utilise quoi ? L'annexe A du RFC contient une mise en œuvre en JavaScript, et l'annexe G une liste d'autres mises en œuvre (que vous pouvez aussi trouver en ligne <https://github.com/cyberphone/json-canonicalization>). Ainsi, en npm, il y a et, en Java, il y a . (Si vous voulez programmer la canonicalisation de JSON vous-même, l'annexe F contient d'utiles conseils aux programmeurs. Mais, ici, on se contentera de logiciels déjà écrits.)

Commençons avec Python, et . Avec cet exemple de départ :

```
{
       "numbers": [333333333.33333329, 1E30, 4.50,
                   2e-3, 0.00000000000000000000000001],
       "string": "\u20ac$\u000F\u000aA'\u0042\u0022\u005c\\\"\/",
       "literals": [null, true, false]
 }
```

Et ce programme :

```
from org.webpki.json.Canonicalize import canonicalize

import json
import sys

raw = open(sys.argv[1]).read()
data = canonicalize(json.loads(raw))
print(data.decode())
```

On obtient :

```
% git clone https://github.com/cyberphone/json-canonicalization.git
% export PYTHONPATH=json-canonicalization/python3/src
% ./canonicalize.py test.json
{"literals":[null,true,false],"numbers":[333333333.3333333,1e+30,4.5,0.002,1e-27],"string":"€$\u000f\nA'B\"\\\\
```

Et en Go ? Avec le même fichier JSON de départ, et ce programme :

```
package main

import ("flag";
"fmt";
"os";)

import "webpki.org/jsoncanonicalizer"

func main() {
flag.Parse()
file, err := os.Open(flag.Arg(0))
if err != nil {
panic(err)
}
data := make([]byte, 1000000)
count, err := file.Read(data)
if err != nil {
panic(err)
}
result, err := jsoncanonicalizer.Transform(data[0:count])
if err != nil {
panic(err);
} else {
fmt.Printf("%s", result);
}
}
```

On arrive au même résultat (ce qui est bien le but de la canonicalisation) :

```
% export GOPATH=$(pwd)/json-canonicalization/go
% go build canonicalize.go
%  ./canonicalize.py test.json
{"literals":[null,true,false],"numbers":[333333333.3333333,1e+30,4.5,0.002,1e-27],"string":"€$\u000f\nA'B\"\\\\
```

Comme notre RFC contraint le JSON à suivre le sous-ensemble I-JSON (RFC 7493), si le texte d'entrée ne l'est pas, il va y avoir un problème. Ici, avec un texte JSON qui n'est **pas** du I-JSON (deux membres de l'objet principal ont le même nom) :

```
% cat test3.json
{
  "foo": 1,
  "bar": 3,
  "foo": 42
}

% ./canonicalize test3.json
panic: Duplicate key: foo
```

(Un autre code en Go est en .)

L'excellent programme jq <https://www.bortzmeyer.org/jq.html> n'a pas d'option explicite pour canonicaliser, mais Sébastien Lecacheur me fait remarquer que les options `-c` et `-S` donnent apparemment le même résultat (y compris sur les nombres, ce qui ne semble pas documenté) :

```
% cat example.json
{
  "foo":                                1,
  "bar": 3000000000000000000000000000000,
"baz": true
}

% jq -cS . example.json
{"bar":3e+30,"baz":true,"foo":1}
```

Par contre, je n'ai rien trouvé sur ce formateur en ligne <https://jsonformatter.org/>.

Comme il y a un brevet pour tout, notez qu'apparemment VMware prétend avoir inventé la canonicalisation JSON (cf. ce signalement <https://datatracker.ietf.org/ipr/3585/>, et le commentaire « *"The patent is effectively a JSON based "remake" of XML's "enveloped" signature scheme which in the submitter's opinion makes it invalid."* »).