Network Working Group
Internet-Draft
Intended status: Informational
Expires: December 27, 2018

A. Rundgren
Independent

June 25, 2018

JSON Canonicalization Scheme (JCS)
draft-rundgren-json-canonicalization-scheme-01

Abstract

Cryptographic operations like hashing and signing depend on that the target data does not change during serialization, transport, or parsing. By applying the rules defined by JCS (JSON Canonicalization Scheme), data provided in the JSON [RFC8259] format can be exchanged "as is", while still being subject to secure cryptographic operations. JCS achieves this by combining the strict serialization of JSON primitives defined in ECMAScript [ES6] with a platform independent sorting scheme.

The intended audiences of this document are JSON tool vendors, as well as designers of JSON based cryptographic solutions.

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

Cryptographic operations like hashing and signing depend on that the target data does not change during serialization, transport, or parsing. A straightforward way of accomplishing this is converting the data into a format which has a simple and fixed representation like Base64Url [RFC4648] which for example have been used in JWS [RFC7515]. Another solution is creating a canonicalized version of the target data with XML Signature [XMLDSIG] as a prime example.

Since the objective was keeping the data "as is", the canonicalization method was selected. For avoiding "reinventing the wheel", JCS relies on serialization of JSON primitives compatible with ECMAScript (aka JavaScript) beginning with version 6 [ES6], from now on simply referred to as "ES6".

Seasoned XML developers recalling difficulties getting signatures to validate (usually due to different interpretations of the quite intricate XML canonicalization rules as well as of the equally extensive Web Services security standards), may rightfully wonder why this particular effort would succeed. The reasons are twofold:

- JSON is a considerably simpler format than XML, as well as lacking support for the powerful (but complex) namespace concept.

- ES6 compatible JSON serialization is already supported by most Web browsers, Node.js [NODEJS], as well as by third party libraries like Open Keystore [OPENKEY], giving the proposed canonicalization scheme a head start. Also see Appendix E.

The JCS specification describes how JSON serializing rules compliant with ES6 combined with an elementary sorting scheme, can be used for supporting "Crypto Safe" JSON.

JCS is compatible with some existing systems relying on JSON canonicalization such as JWK Thumbprint [RFC7638] and Keybase [KEYBASE].

2. Terminology

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

3. Detailed Operation

This section describes the different issues related to JSON canonicalization, and how they are addressed by JCS.

3.1. Creation of JSON Data

In order to canonicalize JSON data, an internal representation of the JSON data is needed. This can be achieved by:

- Parsing externally supplied JSON data.

o Programmatic creation of JSON data.

Irrespective of method used, the JSON data MUST be compatible both with ES6 and I-JSON [RFC7493], which implies the following:

o There MUST NOT be any duplicate property names within an "Object".

o Data of the type "String" MUST be expressible as Unicode [UNICODE] strings. Also see Section 3.2.2.2.

o Data of the type "Number" MUST be expressible as IEEE-754 [IEEE754] double precision values. Also see Section 3.2.2.3.

3.2. Canonicalization of JSON Data

The following sub sections describe the steps required for creating a canonicalized version of internal JSON data elaborated on in the previous section.

Appendix A shows sample code for an ES6 based canonicalizer, matching the JCS specification.

3.2.1. Whitespace Handling

Possible whitespace between JSON elements MUST be ignored (not emitted).

3.2.2. Serialization of Primitive Data Types

Assume that you parse a JSON object like the following:

```
{
  "numbers": [333333333.33333329, 1E30, 4.50,
              2e-3, 0.000000000000000000000000001],
  "string": "\u20ac$\u000F\u000aA'\u0042\u0022\u005c\\\"\/",
  "literals": [null, true, false]
}
```

If you subsequently serialize the object created by the operation above using an serializer compliant with ES6's "JSON.stringify()", the result would (with a line wrap added for display purposes only), be rather divergent with respect to representation of data:

```
{"numbers":[333333333.3333333,1e+30,4.5,0.002,1e-27],"string":
"\u20ac$\u000f\nA'B\"\\\\\"/","literals":[null,true,false]}
```

Note: \u20ac denotes the Euro character, which not being ASCII, is currently not displayable in RFCs.

The reason for the difference between the parsed data and its serialized counterpart, is due to a wide tolerance on input data (as defined by JSON [RFC8259]), while output data (as defined by ES6), has a fixed representation. As can be seen by the example, numbers are subject to rounding as well.

The following sub sections describe serialization of primitive JSON data types according to JCS. This part is identical to that of ES6.

3.2.2.1. Serialization of Literals

The JSON literals "null", "true", and "false" present no challenge since they already have a fixed definition in JSON [RFC8259].

3.2.2.2. Serialization of Strings

For JSON data of the type "String" (which includes "Object" property names as well), each character MUST be serialized as described in Section 24.3.2.2 of ES6.

If the Unicode value falls within the traditional ASCII control character range (U+0000 through U+001F), it MUST be serialized using lowercase hexadecimal Unicode notation (\uhhhh) unless it is in the set of predefined JSON control characters U+0008, U+0009, U+000A, U+000C or U+000D which MUST be serialized as \b, \t, \n, \f and \r respectively.

If the Unicode value is outside of the ASCII control character range, it MUST be serialized "as is" unless it is equivalent to U+005C (\) or U+0022 (") which MUST be serialized as \\ and \" respectively.

Finally, the serialized string value MUST be enclosed in double quotes (").

Note that many JSON systems permit the use of invalid Unicode data like "lone surrogates" (e.g. U+DEAD), which also is dealt with in a platform specific way. Since this leads to interoperability issues including broken signatures, such usages MUST be avoided.

Note that although the Unicode standard offers a possibility combining certain characters into one, referred to as "Unicode Normalization" (https://www.unicode.org/reports/tr15/ [1]), such functionality MUST be delegated to the application layer which already is the case for most other uses of JSON.

3.2.2.3. Serialization of Numbers

JSON data of the type "Number" MUST be serialized according to Section 7.1.12.1 of ES6; for maximum interoperability preferably including the "Note 2" enhancement as well. The latter is implemented by for example Google's V8 [V8].

Due to the relative complexity of this part, it is not included in this specification.

Note that ES6 builds on the IEEE-754 [IEEE754] double precision standard for storing "Number" data. Appendix B holds a set of IEEE-754 sample values and their corresponding JSON serialization.

Occasionally applications need higher precision or longer integers than offered by the current implementation of JSON "Number" in ES6. Appendix D outlines how this can be achieved in a portable and extensible way.

3.2.3. Sorting of Object Properties

Although the previous step indeed normalized the representation of primitive JSON data types, the result would not qualify as "canonicalized" since "Object" properties are not in lexicographic (alphabetical) order.

Applied to the sample in Section 3.2.2, a properly canonicalized version should (with a line wrap added for display purposes only), read as:

```
{"literals":[null,true,false],"numbers":[333333333.3333333,
1e+30,4.5,0.002,1e-27],"string":"\u20ac$\u000f\nA'B\"\\\\\"/"}
```

Note: \u20ac denotes the Euro character, which not being ASCII, is currently not displayable in RFCs.

The rules for lexicographic sorting of JSON properties according to JCS are as follows:

- "Object" properties are sorted in a recursive manner which means that a found JSON child "Object" type MUST be subject to sorting as well.

- JSON "Array" data MUST also be checked for the presence of sortable JSON "Object" elements, but array element order MUST NOT be changed.

When a JSON "Object" is about to have its properties sorted, the following measures MUST be adhered to:

- o The sorting process is applied to the internal representation of property strings. That is, their state before serialization.

- o Property strings to be sorted depend on that strings are internally represented as arrays of 16-bit unsigned integers where each integer holds a single UCS2/UTF-16 [UNICODE] code unit. The sorting is based on pure value comparisons, independent of locale settings.

- o Property strings either have different values at some index that is a valid index for both strings, or their lengths are different, or both. If they have different values at one or more index positions, let k be the smallest such index; then the string whose value at position k has the smaller value, as determined by using the < operator, lexicographically precedes the other string. If there is no index position at which they differ, then the shorter string lexicographically precedes the longer string.

The rationale for basing the sort algorithm on UCS2/UTF-16 code units is that it maps directly to the string type in ECMAScript, Java and .NET. Systems using another representation of string data will need to convert JSON property strings into arrays of UCS2/UTF-16 code units before sorting.

Note: for the purpose obtaining a deterministic property order, sorting on UTF-8 or UTF-32 encoded data would also work, but the result would differ (and thus be incompatible with this specification).

3.2.4. UTF-8 Generation

Finally, in order to create a platform independent representation, the resulting JSON string data MUST be encoded in UTF-8.

Applied to the sample in Section 3.2.3 this should yield the following bytes here shown in hexadecimal notation:

```
7b 22 6c 69 74 65 72 61 6c 73 22 3a 5b 6e 75 6c 6c 2c 74 72
75 65 2c 66 61 6c 73 65 5d 2c 22 6e 75 6d 62 65 72 73 22 3a
5b 33 33 33 33 33 33 33 33 33 2e 33 33 33 33 33 33 33 2c 31
65 2b 33 30 2c 34 2e 35 2c 30 2e 30 30 32 2c 31 65 2d 32 37
5d 2c 22 73 74 72 69 6e 67 22 3a 22 e2 82 ac 24 5c 75 30 30
30 66 5c 6e 41 27 42 5c 22 5c 5c 5c 5c 5c 22 2f 22 7d
```

This data is intended to be usable as input to cryptographic functions.

For other uses see Appendix C.

4. IANA Considerations

This document has no IANA actions.

5. Security Considerations

JSON parsers MUST check that input data conforms to the JSON [RFC8259] specification.

6. Acknowledgements

Building on ES6 "Number" normalization was originally proposed by James Manger. This ultimately led to the adoption of the entire ES6 serialization scheme for JSON primitives.

Other people who have contributed with valuable input to this specification include Mike Jones, Mike Miller, Mike Samuel, Michal Wadas, Richard Gibson and Scott Ananian.

7. References

7.1. Normative References

[ES6] Ecma International, "ECMAScript 2015 Language Specification", <https://www.ecma-international.org/ecma-262/6.0/index.html>.

[IEEE754] IEEE, "IEEE Standard for Floating-Point Arithmetic", August 2008, <http://grouper.ieee.org/groups/754/>.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[RFC7493] Bray, T., Ed., "The I-JSON Message Format", RFC 7493, DOI 10.17487/RFC7493, March 2015, <https://www.rfc-editor.org/info/rfc7493>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/info/rfc8174>.

[RFC8259] Bray, T., Ed., "The JavaScript Object Notation (JSON) Data Interchange Format", STD 90, RFC 8259, DOI 10.17487/RFC8259, December 2017, <https://www.rfc-editor.org/info/rfc8259>.

[UNICODE] The Unicode Consortium, "The Unicode Standard, Version 10.0.0", <https://www.unicode.org/versions/Unicode10.0.0/>.

7.2. Informal References

[KEYBASE] "Keybase", <https://keybase.io/docs/api/1.0/canonical_packings#json>.

[NODEJS] "Node.js", <https://nodejs.org>.

[OPENAPI] "The OpenAPI Initiative", <https://www.openapis.org/>.

[OPENKEY] "Open Keystore", <https://github.com/cyberphone/openkeystore>.

[RFC4648] Josefsson, S., "The Base16, Base32, and Base64 Data Encodings", RFC 4648, DOI 10.17487/RFC4648, October 2006, <https://www.rfc-editor.org/info/rfc4648>.

[RFC7515] Jones, M., Bradley, J., and N. Sakimura, "JSON Web Signature (JWS)", RFC 7515, DOI 10.17487/RFC7515, May 2015, <https://www.rfc-editor.org/info/rfc7515>.

[RFC7638] Jones, M. and N. Sakimura, "JSON Web Key (JWK) Thumbprint", RFC 7638, DOI 10.17487/RFC7638, September 2015, <https://www.rfc-editor.org/info/rfc7638>.

[V8] Google LLC, "Chrome V8 Open Source JavaScript Engine", <https://developers.google.com/v8/>.

[XMLDSIG] W3C, "XML Signature Syntax and Processing Version 1.1", <https://www.w3.org/TR/xmldsig-core1/>.

7.3. URIs

[1] https://www.unicode.org/reports/tr15/

[2] https://tools.ietf.org/html/draft-staykov-hu-json-canonical-form-00

[3] https://gibson042.github.io/canonicaljson-spec/

[4] https://www.npmjs.com/package/canonicalize

[5] http://wiki.laptop.org/go/Canonical_JSON

[6] https://github.com/cyberphone/json-canonicalization

Appendix A. ES6 Sample Canonicalizer

Below is a functionally complete example of a JCS compliant canonicalizer for usage with ES6 based systems.

Note: The primary purpose of this code is highlighting the canonicalization algorithm. Using the full power of ES6 would reduce the code size considerably but would also be more difficult to follow by non-experts.

```
var canonicalize = function(object) {

    var buffer = '';
    serialize(object);
    return buffer;

    function serialize(object) {
        if (object === null || typeof object !== 'object') {
            /////////////////////////////////////////////////
            // Primitive data type - Use ES6/JSON          //
            /////////////////////////////////////////////////
            buffer += JSON.stringify(object);

        } else if (Array.isArray(object)) {
            /////////////////////////////////////////////////
            // Array - Maintain element order              //
            /////////////////////////////////////////////////
            buffer += '[';
            let next = false;
            object.forEach((element) => {
                if (next) {
                    buffer += ',';
                }
                next = true;
                /////////////////////////////////////////
                // Array element - Recursive expansion //
                /////////////////////////////////////////
                serialize(element);
            });
            buffer += ']';

        } else {
```

```
                ////////////////////////////////////////////////
                // Object - Sort properties before serializing //
                ////////////////////////////////////////////////
                buffer += '{';
                let next = false;
                Object.keys(object).sort().forEach((property) => {
                    if (next) {
                        buffer += ',';
                    }
                    next = true;
                    ///////////////////////////////////////////////
                    // Property names are strings - Use ES6/JSON //
                    ///////////////////////////////////////////////
                    buffer += JSON.stringify(property);
                    buffer += ':';
                    //////////////////////////////////////////
                    // Property value - Recursive expansion //
                    //////////////////////////////////////////
                    serialize(object[property]);
                });
                buffer += '}';
            }
        }
    };
```

Appendix B. Number Serialization Samples

The following table holds a set of ES6 "Number" serialization samples, including some edge cases. The column "ES6 Internal" refers to the internal ES6 representation of the "Number" data type which is based on the IEEE-754 [IEEE754] standard using 64-bit (double precision) values, here expressed in hexadecimal.

ES6 Internal	JSON Representation	Comment
0000000000000000	0	Zero
8000000000000000	0	Minus zero
0000000000000001	5e-324	Smallest pos number
8000000000000001	-5e-324	Smallest neg number
7fefffffffffffff	1.7976931348623157e+308	Largest pos number
ffefffffffffffff	-1.7976931348623157e+308	Largest neg number
4340000000000000	9007199254740992	Largest pos integer
c340000000000000	-9007199254740992	Largest neg integer
7fffffffffffffff		Error (NaN)
7ff0000000000000		Error (Infinity)
44b52d02c7e14af5	9.999999999999997e+22	
44b52d02c7e14af6	1e+23	
44b52d02c7e14af7	1.0000000000000001e+23	
444b1ae4d6e2ef4e	999999999999999700000	
444b1ae4d6e2ef4f	999999999999999900000	
444b1ae4d6e2ef50	1e+21	
444b1ae4d6e2ef51	1.0000000000000001e+21	
41b3de4355555553	333333333.3333332	
41b3de4355555554	333333333.33333325	
41b3de4355555555	333333333.3333333	
41b3de4355555556	333333333.3333334	
41b3de4355555557	333333333.33333343	

Note: for maximum compliance with ECMAScript's JSON object, values that are to be interpreted as true integers, SHOULD be in the range -9007199254740991 to 9007199254740991.

Appendix C. Canonicalized JSON as "Wire Format"

Since the result from the canonicalization process (see Section 3.2.4), is fully valid JSON, it can also be used as "Wire Format". However, this is just an option since cryptographic schemes based on JCS, in most cases would not depend on that externally supplied JSON data already is canonicalized.

In fact, the ES6 standard way of serializing objects using "JSON.stringify()" produces a more "logical" format, where properties are kept in the order they were created or received. The example below shows an address record which could benefit from ES6 standard serialization:

```
{
  "name": "John Doe",
  "address": "2000 Sunset Boulevard",
  "city": "Los Angeles",
  "zip": "90001",
  "state": "CA"
}
```

Using canonicalization the properties above would be output in the order "address", "city", "name", "state" and "zip", which adds fuzziness to the data from a human (developer or technical support), perspective.

That is, for many applications, canonicalization would only be used internally for creating a "hashable" representation of the data needed for cryptographic operations.

Note that if message size is not a concern, you may even send "Pretty Printed" JSON data on the wire (since whitespace always is ignored by the canonicalization process).

Appendix D. Dealing with Big Numbers

There are two major issues associated with the JSON "Number" type, here illustrated by the following sample object:

```
{
  "giantNumber": 1.4e+9999,
  "payMeThis": 26000.33,
  "int64Max": 9223372036854775807
}
```

Although the sample above conforms to JSON (according to [RFC8259]), there are some practical hurdles to consider:

- Standard JSON parsers rarely process "giantNumber" in a meaningful way. 64-bit integers like "int64Max" normally pass through parsers, but in systems like ES6, at the expense of lost precision.

- Another issue is that parsers typically would use different schemes for handling "giantNumber" and "int64Max". In addition, monetary data like "payMeThis" would presumably not rely on a floating point system due to rounding issues with respect to decimal arithmetic.

  The (to the author NB), only known way handling this kind of "overloading" of the "Number" type (at least in an extensible manner), is through mapping mechanisms, instructing parsers what to do with different properties based on their name. However, this greatly limits the value of using the "Number" type outside of its original somewhat constrained, JavaScript context.

For usage with JCS (and in fact for any usage of JSON by multiple parties potentially using independently developed software), numbers that do not have a natural place in the current JSON ecosystem MUST be wrapped using the JSON "String" type. This is close to a de-facto standard for open systems.

Aided by a mapping system; be it programmatic like

```
var obj = JSON.parse('{"giantNumber": "1.4e+9999"}');
var biggie = new BigNumber(obj.giantNumber);
```

or declarative schemes like OpenAPI [OPENAPI], there are no real limits, not even when using ES6.

Appendix E. Implementation Guidelines

The optimal solution is integrating support for JCS directly in JSON parsers and serializers. However, this is not always realistic. Fortunately JCS support can be performed through externally supplied canonicalizer software, enabling signature creation schemes like the following:

1. Create the data to be signed.

2. Serialize the data using existing JSON tools.

3. Let the external canonicalizer process the serialized data and return canonicalized result data.

4. Sign the canonicalized data.

5. Add the resulting signature value to the original JSON data through a designated signature property.

6. Serialize the completed (now signed) JSON object using existing JSON tools.

A compatible signature verification scheme would then be as follows:

1. Parse the signed JSON data using existing JSON tools.

2. Read and save the signature value from the designated signature property.

3. Remove the signature property from the parsed JSON object.

4. Serialize the remaining JSON data using existing JSON tools.

5. Let the external canonicalizer process the serialized data and return canonicalized result data.

6. Verify that the canonicalized data matches the saved signature value using the algorithm and key used for creating the signature.

A canonicalizer like above is effectively only a "filter", potentially usable with a multitude of quite different cryptographic schemes.

Using an integrated canonicalizer, you would eliminate the serialization and parsing step before the canonicalization, for both processes. That is, canonicalization would typically be an additional "mode" for a JSON serializer.

Appendix F. Other JSON Canonicalization Efforts

There are (and have been) other efforts creating "Canonical JSON". Below is a list of URLs to some of them:

o https://tools.ietf.org/html/draft-staykov-hu-json-canonical-form-00 [2]

o https://gibson042.github.io/canonicaljson-spec/ [3]

o https://www.npmjs.com/package/canonicalize [4]

o http://wiki.laptop.org/go/Canonical_JSON [5]

Appendix G. Development Portal

The JSC specification is currently developed at https://github.com/cyberphone/json-canonicalization [6].

The portal also provides software for testing.

Author's Address

Anders Rundgren
Independent
Montpellier
France

Email: anders.rundgren.net@gmail.com
URI: https://www.linkedin.com/in/andersrundgren/