Network Working Group
Internet-Draft
Intended status: Standards Track
Expires: July 28, 2019

A. Rundgren
Independent
B. Jordan
Symantec Corporation
S. Erdtman
Spotify AB

January 24, 2019

JSON Canonicalization Scheme (JCS)
draft-rundgren-json-canonicalization-scheme-03

Abstract

Cryptographic operations like hashing and signing requires that the original data does not change during serialization or parsing. By applying the rules defined by the JSON Canonicalization Scheme (JCS), data provided in JSON [RFC8259] format can be exchanged "as is", while still being usable by secure cryptographic operations. JCS achieves this by building on the serialization formats for JSON primitives as defined by ECMAScript [ES6], constraining JSON data to the I-JSON [RFC7493] subset, and through a platform independent property 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

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on July 28, 2019.

Table of Contents

1. Introduction

Cryptographic operations like hashing and signing requires that the original data does not change during serialization or parsing. One way of accomplishing this is converting the data into a format that has a simple and fixed representation like Base64Url [RFC4648], which is how JWS [RFC7515] addressed this issue.

Another solution is to create a canonical version of the data, similar to what was done for the XML Signature [XMLDSIG] standard. The primary advantage with a canonicalizing scheme is that data can be kept in its original form. This is the core rationale behind JCS. Put another way: by using canonicalization a JSON Object may remain a JSON Object even after being signed which simplifies system design, documentation and logging.

To avoid "reinventing the wheel", JCS relies on serialization of JSON primitives compatible with ECMAScript (aka JavaScript) beginning with version 6 [ES6], hereafter 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 JCS would not suffer from similar issues. The reasons are twofold:

- The absence of a namespace concept and default values, as well as constraining data to the I-JSON subset eliminate the need for specific parsing schemes for dealing with canonicalization.

- JCS compatible serialization of JSON primitives is supported by most current Web browsers and as well as by Node.js [NODEJS], while the full JCS specification is supported by multiple Open Source implementations (see Appendix F). See also Appendix E.

In summary, the JCS specification describes how serialization of JSON primitives compliant with ES6, combined with an elementary property 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 creating a canonical JSON representation, and how they are addressed by JCS.

3.1. Creation of Input Data

In order to serialize JSON data, one needs data that is adapted for JSON serialization. This is usually achieved by:

- Parsing previously generated JSON data.
- Programmatically creating data.

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

- There MUST NOT be any duplicate property names within an element to be serialized as a JSON Object.
- String data MUST be expressible as Unicode [UNICODE]. For more information see Section 3.2.2.2.
- Numeric data MUST be expressible as IEEE-754 [IEEE754] double precision values. For more information see Section 3.2.2.3.

3.2. Generation of Canonical JSON Data

The following subsections describe the steps required for creating a canonical JSON representation of the 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

Whitespace between JSON elements MUST NOT be 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 parsed data using a 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 subsections 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 String data (which includes JSON Object property names as well), each character MUST be serialized as described below (also matching 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: some 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: 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.

3.2.2.3. Serialization of Numbers

JSON data of type Number MUST be serialized according to Section 7.1.12.1 of [ES6] including the "Note 2" enhancement.

Due to the relative complexity of this part, the algorithm itself is not included in this document. However, the specification is fully implemented by for example Google's V8 [V8]. The open source Java implementation mentioned in Appendix F uses a recently developed number serialization algorithm called Ryu [RYU].

ES6 builds on the IEEE-754 [IEEE754] double precision standard for representing JSON 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 IEEE-754 double precision. 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 "canonical" since JSON 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 Object properties according to JCS are as follows:

- JSON Object properties MUST be sorted in a recursive manner which means that possible JSON child Objects MUST have their properties sorted as well.

- JSON Array data MUST also be scanned for presence of JSON Objects (and applying associated property sorting), 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:

- The sorting process is applied to property strings in their "raw" (unescaped) form. That is, a newline character is treated as U+000A.

- Property strings to be sorted are formatted as arrays of UTF-16 [UNICODE] code units. The sorting is based on pure value comparisons, where code units are treated as unsigned integers, independent of locale settings.

- 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.

  In plain english this means that property names are sorted in ascending order like the following:

  ```
  ""
  "a"
  "aa"
  "ab"
  ```

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

units before sorting. The conversion from UTF-8 or UTF-32 to UTF-16 is defined by the Unicode [UNICODE] standard.

Note: for the purpose of 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 methods as well as for value comparisons of JSON objects.

For other uses see Appendix C.

4. IANA Considerations

This document has no IANA actions.

5. Security Considerations

It is vital performing "sanity" checks on input data to avoid overflowing buffers and similar things that could affect the integrity of the system.

6. Acknowledgements

Building on ES6 Number serialization 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 Bron Gondwana, John-Mark Gurney, Mike Jones, Mike Miller, Mike Samuel, Michal Wadas, Richard Gibson, Robert Tupelo-Schneck 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/>.

[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>.

[RYU] Ulf Adams, "Ryu floating point number serializing algorithm", <https://github.com/ulfjack/ryu>.

[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://www.npmjs.com/package/canonicalize

[3] https://github.com/erdtman/java-json-canonicalization

[4] https://github.com/cyberphone/json-canonicalization/tree/master/go

[5] https://github.com/cyberphone/json-canonicalization/tree/master/dotnet

[6] https://github.com/cyberphone/json-canonicalization/tree/master/python3

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

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

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

[10] https://github.com/cyberphone/ietf-json-canon

[11] https://cyberphone.github.io/ietf-json-canon

[12] 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 compatible Number serialization samples, including some edge cases. The column "IEEE-754" 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.

IEEE-754	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	9999999999999700000	
444b1ae4d6e2ef4f	9999999999999900000	
3eb0c6f7a0b5ed8d	0.000001	
3eb0c6f7a0b5ed8c	9.999999999999997e-7	
41b3de4355555553	333333333.3333332	
41b3de4355555554	333333333.33333325	
41b3de4355555555	333333333.3333333	
41b3de4355555556	333333333.3333334	
41b3de4355555557	333333333.33333343	
becbf647612f3696	-0.000003333333333333333	

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.

Note: although a set of specific integers like 2**68 (4430000000000000 in IEEE-754 format) could be regarded as having extended precision, the JCS/ES6 number serialization algorithm does not take this in consideration.

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: 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 applications would normally use different data types for "giantNumber" and "int64Max". In addition, monetary data like "payMeThis" would presumably not rely on

floating point data types due to rounding issues with respect to decimal arithmetic.

The established way handling this kind of "overloading" of the JSON 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 serializers (parsers need no changes). That is, canonicalization would just be an additional "mode" for a JSON serializer. However, this is currently not the case. 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 a JSON serializer with integrated JCS support, the serialization performed before the canonicalization step could be eliminated for both processes.

Appendix F. Open Source Implementations

The following Open Source implementations have been verified to be compatible with JCS:

- JavaScript: https://www.npmjs.com/package/canonicalize [2]

- Java: https://github.com/erdtman/java-json-canonicalization [3]

- Go: https://github.com/cyberphone/json-canonicalization/tree/master/go [4]

- .NET/C#: https://github.com/cyberphone/json-canonicalization/tree/master/dotnet [5]

- Python: https://github.com/cyberphone/json-canonicalization/tree/master/python3 [6]

Appendix G. Other JSON Canonicalization Efforts

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

- https://tools.ietf.org/html/draft-staykov-hu-json-canonical-form-00 [7]
- https://gibson042.github.io/canonicaljson-spec/ [8]
- http://wiki.laptop.org/go/Canonical_JSON [9]

Appendix H. Development Portal

The JCS specification is currently developed at: https://github.com/cyberphone/ietf-json-canon [10].

The most recent "editors' copy" can be found at: https://cyberphone.github.io/ietf-json-canon [11].

JCS source code and test data is available at: https://github.com/cyberphone/json-canonicalization [12]

Authors' Addresses

Anders Rundgren
Independent
Montpellier
France

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

Bret Jordan
Symantec Corporation
350 Ellis Street
Mountain View CA 94043
USA

Email: bret_jordan@symantec.com

Samuel Erdtman
Spotify AB
Birger Jarlsgatan 61, 4tr
Stockholm 113 56
Sweden

Email: erdtman@spotify.com