Workgroup: Network Working Group
Published: 20 November 2020
Intended Status: Informational
Expires: 24 May 2021
Authors: B. Jordan, Ed. S. Erdtman A. Rundgren
Broadcom Spotify AB Independent

JWS Clear Text JSON Signature Option (JWS/CT)

Abstract

This document describes a method for extending the scope of the JSON Web Signature (JWS) standard, called JWS/CT. By combining the detached mode of JWS with the JSON Canonicalization Scheme (JCS), JWS/CT enables JSON objects to remain in the JSON format after being signed (aka "Clear Text" signing).

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

This specification introduces a method for augmenting data expressed in the JSON [RFC8259] notation, with enveloped signatures, similar to the scheme used in the XML Signature [XMLDSIG] standard. For interoperability reasons this specification constrains JSON objects to the I-JSON [RFC7493] subset.

To avoid "reinventing the wheel", this specification leverages the JSON Web Signature (JWS) [RFC7515] standard.

By building on the detached mode of JWS in combination with the JSON Canonicalizion Scheme (JCS) [RFC8785], JSON objects to be signed can be kept in the JSON format. This arrangement is here referred to as JWS/CT, where CT stands for "Clear Text" signing.

The primary motivations for keeping signed JSON objects in the JSON format include *simplified documentation*, *debugging*, and *logging*, as well as for maintaining a *consistent message structure*.

Another target is HTTP-based signature schemes that currently utilize HTTP header values for holding detached signatures. By rather using the method described herein, signed JSON-formatted HTTP requests and responses may be self-contained and thus be *serializable*. The latter facilitates such data to be

* stored in databases
* passed through intermediaries
* embedded in other JSON objects
* counter-signed

without losing the ability to (at any time) verify signatures.

Appendix B outlines different ways to handle multiple signatures including counter-signing using JWS/CT.

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

2. Terminology

Note that this document is not on the IETF standards track. However, a conformant implementation is supposed to adhere to the specified behavior for security and interoperability reasons. This text uses BCP 14 to describe that necessary behavior.

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 details related to signing and validating signatures based on this specification.

The following characteristics are vital to know for prospective JWS/CT implementers and users:

* With the exception of the reliance on the detached mode described in Appendix F of JWS, JWS/CT does not alter the JWS signature creation process, validation process, or format. This means that the contents of JWS headers as well as things related to signature algorithms and cryptographic keys are out of scope for this specification.

* JWS/CT exclusively depends on the JWS *Compact Serialization* mode.

* JCS [RFC8785] constrains JSON objects to the I-JSON [RFC7493] subset.

The signature creation and signature validation sections feature samples using the HS256 JOSE algorithm [RFC7518] with a 256-bit key having the following value here expressed as hexadecimal bytes:

```
7f dd 85 1a 3b 9d 2d af c5 f0 d0 00 30 e2 2b 93 43 90 0c d4 2e de
49 48 56 8a 4a 2e e6 55 29 1a
```

3.1. Signature Creation

The following sub-sections describe how JSON objects can be signed according to the JWS/CT specification.

3.1.1. Create the JSON Object to be Signed

Create or parse the JSON object to be signed.

For illustrating the subsequent operations the following sample object is used:

```
{
  "statement": "Hello signed world!",
  "otherProperties": [2000, true]
}
```

3.1.2. Canonicalize the JSON Object to be Signed

Use the result of the previous step as input to the canonicalization process described in JCS [RFC8785].

Applied to the sample, the following JSON string should be generated:

```
{"otherProperties":[2000,true],"statement":"Hello signed world!"}
```

After encoding the string above in the UTF-8 [UNICODE] format, the following bytes (here in hexadecimal notation) should be generated:

```
7b 22 6f 74 68 65 72 50 72 6f 70 65 72 74 69 65 73 22 3a 5b 32 30
30 30 2c 74 72 75 65 5d 2c 22 73 74 61 74 65 6d 65 6e 74 22 3a 22
48 65 6c 6c 6f 20 73 69 67 6e 65 64 20 77 6f 72 6c 64 21 22 7d
```

3.1.3. Generate a JWS String

Use the result of the previous step as JWS Payload to the signature process described in https://tools.ietf.org/html/rfc7515#appendix-F of JWS.

Note: although it is true that using the "Unencoded Payload" mode of RFC7797 [RFC7797] would eliminate an *internal-only*, base64url encoding step, the performance hit should be marginal for most real-world applications. The current design also makes JWS/CT independent of JWS library support for the "b64":false and "crit":["b64"] JWS header items required by RFC7797. However, this specification does not in any way prohibit the use RFC7797, it rather leaves it as an *implementer option*.

For the sample, the JWS header is assumed to be:

```
{"alg":"HS256"}
```

The resulting JWS string should then after payload removal and using the key specified in Section 3, read as follows:

eyJhbGciOiJIUzI1NiJ9..VHVItCBCb8Q5CI-49imarDtJeSxH2uLU0DhqQP5Zjw4

3.1.4. Assemble the Signed JSON Object

Before a complete signed object can be created, a dedicated top-level property for holding the JWS signature string needs to be defined. The only requirement is that this property **MUST NOT** clash with any other top-level property name. The JWS string itself **MUST** be supplied as a JSON string argument ("") to the signature property.

For the sample, the property name "signature" is assumed to be the designated holder of the JWS string. Equipped with a signature property, the JWS string from the previous section, and the original JSON sample, the process above should result in the following, now signed JSON object (with line breaks in the "signature" property for display purposes only):

```
{
  "statement": "Hello signed world!",
  "otherProperties": [2000, true],
  "signature": "eyJhbGciOiJIUzI1NiJ9..VHVItCBCb8Q5CI-49imar
DtJeSxH2uLU0DhqQP5Zjw4"
}
```

Note: one could equally well apply the signature to the canonicalized version of the JSON object. However, the rearrangement of properties (performed by JCS), may sometimes be considered an issue from a "human" point of view, while computing-wise the order of JSON properties has no impact on the outcome.

3.2. Signature Validation

The following sub-sections describe how JSON objects signed according to the JWS/CT specification can be validated.

3.2.1. Parse the Signed JSON Object

Parse the JSON object that is anticipated to be signed. If the parsing is unsuccessful, the operation **MUST** cause a compliant implementation to terminate with an appropriate error.

To illustrate the subsequent operations the signed JSON object featured in Section 3.1.4 is used as sample.

3.2.2. Fetch the Signature Property String

After successful parsing, retrieve the designated JSON top-level property holding the JWS string. If the property is missing or its argument is not a JSON string (""), the operation **MUST** cause a compliant implementation to terminate with an appropriate error.

For the sample, where the property named "signature" is assumed to hold the JWS string, the operation above should return the following string:

eyJhbGciOiJIUzI1NiJ9..VHVItCBCb8Q5CI-49imarDtJeSxH2uLU0DhqQP5Zjw4

3.2.3. Remove the Signature Property String

Since the signature is calculated over the actual JSON object data, the designated signature property and its argument **MUST** be removed from the signed JSON object.

If applied to the sample the resulting JSON object should read as follows:


```
{
  "statement": "Hello signed world!",
  "otherProperties": [2000, true]
}
```

Note: JSON tools usually by default remove whitespace. In addition, the original ordering of properties may not always be honored. However, none of this has (due to the canonicalization performed by JCS), any impact on the result.

3.2.4. Canonicalize the Remaining JSON Object

Use the result of the previous step as input to the canonicalization process described in JCS [RFC8785].

If applied to the sample the result of the process above should read as follows:


```
{"otherProperties":[2000,true],"statement":"Hello signed world!"}
```


After encoding the string above in the UTF-8 [UNICODE] format, the following bytes (here in hexadecimal notation) should be generated:


```
7b 22 6f 74 68 65 72 50 72 6f 70 65 72 74 69 65 73 22 3a 5b 32 30
30 30 2c 74 72 75 65 5d 2c 22 73 74 61 74 65 6d 65 6e 74 22 3a 22
48 65 6c 6c 6f 20 73 69 67 6e 65 64 20 77 6f 72 6c 64 21 22 7d
```


3.2.5. Validate the JWS String

After extracting the detached mode JWS string and canonicalizing the JSON object (to retrieve the JWS Payload), the JWS string **MUST** be restored as described in https://tools.ietf.org/html/rfc7515#appendix-F of JWS [RFC7515]. The actual JWS validation procedure is not specified here because it is covered by [RFC7515] and also depends on application-specific policies like:

- *Accepted JWS signature algorithms
- *Accepted and/or required JWS header elements
- *Signature key lookup methods

If the validation process for some reason fails, the operation **MUST** cause a compliant implementation to terminate with an appropriate error.

For the sample, validation is straightforward since both the algorithm and the key to use are predefined (see Section 3). The input string to a JWS validator should after the process step above read as follows (with line breaks for display purposes only):

eyJhbGciOiJIUzI1NiJ9.eyJvdGhlclByb3BlcnRpZXMiOlsyMDAwLHRydWVdLCJzdGF0
ZW1lbnQiOiJIZWxsbyBzaWduZWQgd29ybGQhIn0.VHVItCBCb8Q5CI-49imarDtJeSxH2
uLU0DhqQP5Zjw4

4. IANA Considerations

This document has no IANA actions.

5. Security Considerations

This specification inherits all the security considerations of JWS [RFC7515] and JCS [RFC8785].

In similarity to any other signature specification, it is crucial that signatures are verified before acting on the signed payload.

For usage in a wider community, the name of the designated signature property becomes a critical factor that **MUST** be documented and communicated. However, in a properly designed system, a faulty or missing signature **MUST** "only" lead to failed operation, and not to a security breach.

6. References

6.1. Normative References

[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>.

[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>.

[RFC7518] Jones, M., "JSON Web Algorithms (JWA)", RFC 7518, DOI 10.17487/RFC7518, May 2015, <https://www.rfc-editor.org/info/rfc7518>.

[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>.

[RFC8785] Rundgren, A., Jordan, B., and S. Erdtman, "JSON Canonicalization Scheme (JCS)", RFC 8785, DOI 10.17487/RFC8785, June 2020, <https://www.rfc-editor.org/info/rfc8785>.

[UNICODE] The Unicode Consortium, "The Unicode Standard", <https://www.unicode.org/versions/latest/>.

6.2. Informative References

[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>.

[RFC7517] Jones, M., "JSON Web Key (JWK)", RFC 7517, DOI 10.17487/RFC7517, May 2015, <https://www.rfc-editor.org/info/rfc7517>.

[RFC7797] Jones, M., "JSON Web Signature (JWS) Unencoded Payload Option", RFC 7797, DOI 10.17487/RFC7797, February 2016, <https://www.rfc-editor.org/info/rfc7797>.

[RFC8037] Liusvaara, I., "CFRG Elliptic Curve Diffie-Hellman (ECDH) and Signatures in JSON Object Signing and Encryption (JOSE)", RFC 8037, DOI 10.17487/RFC8037, January 2017, <https://www.rfc-editor.org/info/rfc8037>.

[SHS]
NIST, "Secure Hash Standard (SHS)", FIPS PUB 180-4, August 2015, <https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.180-4.pdf>.

[XMLDSIG] W3C, "XML Signature Syntax and Processing Version 1.1", W3C Recommendation, April 2013, <https://www.w3.org/TR/xmldsig-core1/>.

Appendix A. Open-Source Implementations

Due to the simplicity of this specification, there is hardly a need for specific support software. However, JCS which is (at the time of writing), a relatively new design, may be fetched as a separate component for multiple platforms. The following open-source implementations have been verified to be compatible with JCS:

* JavaScript: <https://www.npmjs.com/package/canonicalize>

* Java: <https://mvnrepository.com/artifact/io.github.erdtman/java-json-canonicalization>

* Go: <https://github.com/cyberphone/json-canonicalization/tree/master/go>

* .NET/C#: <https://github.com/cyberphone/json-canonicalization/tree/master/dotnet>

* Python: <https://github.com/cyberphone/json-canonicalization/tree/master/python3>

Appendix B. JWS/CT Application Notes

The following application notes are not a part of the JWS/CT core; they show how JWS/CT can be used in contexts involving multiple signatures.

B.1. Counter Signatures

Consider the following JWS/CT object showing an imaginary real estate business record:

```
{
  "gps": [38.89768255588178, -77.03658644893932],
  "object": {"type": "house", "price": "$635,000"},
  "role": "buyer",
  "name": "John Smith",
  "timeStamp": "2020-11-08T13:56:08Z",
  "signature": "<<Buyer JWS Signature>>"
}
```

Adding a notary signature on top of this could be performed by embedding the former object as follows:

```
{
  "attesting": {
    "gps": [38.89768255588178, -77.03658644893932],
    "object": {"type": "house", "price": "$635,000"},
    "role": "buyer",
    "name": "John Smith",
    "timeStamp": "2020-11-08T13:56:08Z",
    "signature": "<<Buyer JWS Signature>>"
  },
  "role": "notary",
  "name": "Carol Lombardi-Jones",
  "timeStamp": "2020-11-08T13:58:42Z",
  "signature": "<<Notary JWS Signature>>"
}
```

A side effect of this arrangement is that the notary's signature signs not only the notary data, but the buyer's data and signature as well. In most cases this way of adding signatures is advantageous since it maintains the actual order of signing events which also cannot be tampered with without invalidating the outermost signature.

Note that all properties above including "signature" are application specific.

B.2. Detached Signatures

In the case the signing entities are "peers" or are unrelated to each other, counter-signatures like described in Appendix B.1 are not applicable since they presume a specific flow. For supporting *independent* or *asynchronous* signers targeting a common document or data object, an imaginable solution is using a scheme where each signer rather calculates a hash of the target document/data and includes the hash with the signer-specific meta data like the following:

```
{

  <<Common Document/Data to Sign...>>

  "signers": [{
    "sha256": "<<Hash of Document/Data to Sign>>",

    <<Signer-related meta data...>>

    "signature": "<<Signer JWS Signature>>"
  },{
    "sha256": "<<Hash of Document/Data to Sign>>",

    <<Signer-related meta data...>>

    "signature": "<<Signer JWS Signature>>"
  }]
}
```

In this case the object to sign would not be limited to JSON; it could for example be a PDF document hosted on a specific URL. Note that the relying party would have to update the structure for each signature received. In some cases a database would probably be more useful for holding individual signatures since a database can cope with any number of signers as well as keeping track of who have actually signed. The latter is crucial for things like international treaties and company board statements.

Note that "signers", "sha256", and "signature" are application specific property names.

The following sample shows a possible signature object (with line breaks in the signature strings for display purposes only):

```
{
  "statement": "Hello signed world!",
  "otherProperties": [2000, true],
  "signers": [{
    "sha256": "n-i0HIBJKELoTicCK9c5nqJ8cYH0znGRcEbYKoQfm70",
    "timeStamp": "2020-11-18T07:45:28Z",
    "name": "Jane Doe",
    "signature": "eyJhbGciOiJIUzI1NiJ9..57zPdGh88IgI9kECb1u3O
Rhjrbe5mZP4wetM2QCoCBM"
  },{
    "sha256": "n-i0HIBJKELoTicCK9c5nqJ8cYH0znGRcEbYKoQfm70",
    "timeStamp": "2020-11-18T08:03:40Z",
    "name": "John Doe",
    "signature": "eyJhbGciOiJFZERTQSJ9..OQLwF9XHtLru0GYMkG-WS
dSdqJkQ-jxTqLJXtV8dqruJe1DVsBLI8ok0IZu8jXibZPow5W1hbBmdYJAYCu5hCA"
  }]
}
```

Notes:

* "Jane Doe" used the sample key from Section 3 while "John Doe" used the sample key specified in Appendix C.

* The "sha256" properties hold base64url-encoded [RFC4648], SHA256-hashes [SHS] of the canonicalized data created in Section 3.1.2.

B.3. Array of Signatures

Another possibility supporting *multiple and independent* signatures is collecting JWS signature strings in a JSON array object according to the following scheme:

```
{

  <<Common Document/Data to Sign...>>

  "<<Signature property>>": ["<<Signature-1>>",
                             "<<Signature-2>>",
                                        .
                             "<<Signature-n>>"]
}
```

Processing would follow Section 3, with the addition that each signature is dealt with individually.

Compared to Appendix B.2, signature arrays imply that possible signer-specific meta-data is supplied as JWS extensions in the associated signature's base64url-encoded header.

By combining the sample used in Section 3 with the test vector in Appendix C, a valid signature array object could be as follows (with line breaks in the signature strings for display purposes only):


```
{
  "statement": "Hello signed world!",
  "otherProperties": [2000, true],
  "signatures": ["eyJhbGciOiJIUzI1NiJ9..VHVItCBCb8Q5CI-49imar
DtJeSxH2uLU0DhqQP5Zjw4",
                 "eyJhbGciOiJFZERTQSJ9..WAyfK782CRkJh4hcP-OQ3
qUYpH6xY3vfFhaRSzNgG5Eu4p54SyTX25-HjNRN8qE5hmMovd8tycp6I9uqRofiBg"]
}
```


Note that "signatures" is not a key word, it was only selected to highlight the fact that there are multiple signatures.

Appendix C. Test Vector Using the Ed25519 Algorithm

This appendix shows how a signed version of the JSON sample object in Section 3.1.1 would look like if applying the Ed25519 algorithm described in RFC 8037 [RFC8037] (with line breaks in the "signature" property for display purposes only):


```
{
  "statement": "Hello signed world!",
  "otherProperties": [2000, true],
  "signature": "eyJhbGciOiJFZERTQSJ9..WAyfK782CRkJh4hcP-OQ3
qUYpH6xY3vfFhaRSzNgG5Eu4p54SyTX25-HjNRN8qE5hmMovd8tycp6I9uqRofiBg"
}
```


The sample above depends on a JWS header holding the algorithm {"alg":"EdDSA"}, and the following private key, here expressed in the JWK [RFC7517] format:


```
{
  "kty": "OKP",
  "crv": "Ed25519",
  "x": "_kms9bkrbpI1lPLoM2j2gKySS-k89TOuyvgC43dX-Mk",
  "d": "0flr-6bXs459f9qwAq20Zs3NizTGIEH5_rTDFoumFV4"
}
```

Acknowledgements

People who have contributed directly and indirectly with valuable input to this specification include Vladimir Dzhuvinov, Freddi Gyara, and Filip Skokan.

Authors' Addresses

Bret Jordan (editor)
Broadcom
1320 Ridder Park Drive
San Jose, CA 95131
United States of America

Email: bret.jordan@broadcom.com

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

Email: erdtman@spotify.com

Anders Rundgren
Independent
Montpellier
France

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