Workgroup: Network Working Group
Published: 21 June 2022
Intended Status: Informational
Expires: 23 December 2022
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) specification, called JWS/CT (JWS "Clear Text"). 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. In addition to supporting a consistent data format, this arrangement also simplifies documentation, debugging, and logging. The ability to embed signed JSON objects in other JSON objects, makes the use of counter-signatures straightforward.

This informational specification has been produced outside the IETF, is not an IETF standard, and does not have IETF consensus. The intended audiences of this document are JSON tool vendors as well as designers of JSON-based cryptographic solutions.

(https://trustee.ietf.org/license-info) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document.

Table of Contents

1. Introduction
2. Terminology
3. Detailed Operation
  3.1. Signature Creation
    3.1.1. Create the JSON Object to be Signed
    3.1.2. Canonicalize the JSON Object to be Signed
    3.1.3. Generate a JWS String
    3.1.4. Assemble the Signed JSON Object
  3.2. Signature Validation
    3.2.1. Parse the Signed JSON Object
    3.2.2. Fetch the Signature Property String
    3.2.3. Remove the Signature Property String
    3.2.4. Canonicalize the Remaining JSON Object
    3.2.5. Validate the JWS String
4. IANA Considerations
5. Security Considerations
6. References
  6.1. Normative References
  6.2. Informative References
Appendix A. Open-Source Implementations
Appendix B. JWS/CT Application Notes
  B.1. Counter-Signatures
  B.2. Detached Signatures
  B.3. Array of Signatures
Appendix C. Test Vector Using the ES256 Algorithm
Appendix D. Enhanced JWS Processing Option
Acknowledgements
Document History
Authors' Addresses

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 XML Signature [XMLDSIG]. For interoperability and security reasons this specification constrains JSON objects to the I-JSON [RFC7493] subset.

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

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 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 crucial 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 [RFC7515], 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. A slightly enhanced processing option is outlined in Appendix D.

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

- JSON data to be signed MUST be supplied as JSON objects. That is, direct signing of JSON arrays or JSON primitives is out of scope for this specification.

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

The signature creation and signature validation sections (Section 3.1 and Section 3.2 respectively), feature examples 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.

The following example object is used to illustrate the operations in the sections that follow:

```
{
  "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 example, 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 Appendix F of JWS [RFC7515].

For the example, 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 example, 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 example, the process above should result in the following, now signed JSON object (with a line break in the "signature" property for display purposes only):

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

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 expected to have been signed. If the parsing is unsuccessful, the operation **MUST** cause a compliant implementation to terminate processing and return an error indication.

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

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 value, the operation **MUST** cause a compliant implementation to terminate processing and return an error indication.

For the example, 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 example 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 example 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 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 processing and return an error indication.

For the example, 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]. Note that strict conformance to I-JSON [RFC7493] is REQUIRED.

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

However, poorly tested software components may also introduce security issues. Consider the following JSON example:

```
{
  "fromAccount": "1234",
  "toAccount": "4567",
  "amount": {
    "value": 100,
    "currency":"USD"
  }
}
```

A non-compliant JCS implementation could return

```
{"amount":{},"fromAccount":"1234","toAccount":"4567"}
```

giving an attacker the ability to change "amount" to whatever it wants. Note though that this attack presumes that the consumer and

producer use implementations broken in the same way, otherwise the signature would not validate.

For usage in a wider community, the name of the designated signature property becomes a critical factor that **MUST** be documented and communicated.

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>.

[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 (with a line break in the "signature" property for display purposes only):

```
{
  "gps": [38.89768255588178, -77.03658644893932],
  "object": {
    "type": "house",
    "price": "$635,000"
  },
  "role": "buyer",
  "name": "John Smith",
  "timeStamp": "2020-11-08T13:56:08Z",
  "signature": "eyJhbGciOiJIUzI1NiJ9..zlPMniQiz4Eie86oK4xo25z
uyW92csiDqyiQrF6R5ug"
}
```

{:

The signature above was created using the example key from Section 3.

Adding a notary signature on top of this could be performed by embedding the former object as follows (with line breaks in the "signature" properties for display purposes only):

```
{
  "attesting": {
    "gps": [38.89768255588178, -77.03658644893932],
    "object": {
      "type": "house",
      "price": "$635,000"
    },
    "role": "buyer",
    "name": "John Smith",
    "timeStamp": "2020-11-08T13:56:08Z",
    "signature": "eyJhbGciOiJIUzI1NiJ9..zlPMniQiz4Eie86oK4xo25z
uyW92csiDqyiQrF6R5ug"
  },
  "role": "notary",
  "name": "Carol Lombardi-Jones",
  "timeStamp": "2020-11-08T13:58:42Z",
  "signature": "eyJhbGciOiJFUzI1NiJ9..AVmJGUWp1JD0pf2j1_UQWXbf-
qj-2RWxOnyAXihd4POKbnjWqqSBmHPNfgMQFH_s5sXHkIOkDZe2nShqEJOEVA"
}
```

{:

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.

The notary's signature was created using the example key from Appendix C.

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 calculates a hash of the target document/data and includes the hash together with 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 although "signers", "sha256", and "signature" are application specific property names, the objects in the "signers" array are assumed to be fully conformant with the JWS/CT specification.

The following example shows a possible detached signature solution (with line breaks in the "signature" properties for display purposes only):


```
{
  "statement": "Hello signed world!",
  "otherProperties": [2000, true],
  "signers": [{
    "sha256": "n-i0HIBJKELoTicCK9c5nqJ8cYH0znGRcEbYKoQfm70",
    "timeStamp": "2020-11-18T07:45:28Z",
    "name": "Alice",
    "signature": "eyJhbGciOiJIUzI1NiJ9..AE7CnzSYsaspE3yrdsAwi
avd3IdWtdAmDE8FRMwYLA8"
  },{
    "sha256": "n-i0HIBJKELoTicCK9c5nqJ8cYH0znGRcEbYKoQfm70",
    "timeStamp": "2020-11-18T08:03:40Z",
    "name": "Bob",
    "signature": "eyJhbGciOiJFUzI1NiJ9..0tNLy0pLcHUjPhhorpKd5
7a8zTPeqlrOjATiSlPQ1vciE99x6mHmow04tPbJS8dqSqO9c4RkKW6jeL4ZyWpXLA"
  }]
}
```


Notes:

* "Alice" used the example key from Section 3 while "Bob" used the example 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.

* This arrangement requires a two-step validation process where each JWS/CT object in the "signers" array is individually validated, as well as having its "sha256" property compared with the actual hash of the canonicalized common data.

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 example 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 "signatures" property for display purposes only):

```
{
  "statement": "Hello signed world!",
  "otherProperties": [2000, true],
  "signatures": ["eyJhbGciOiJIUzI1NiJ9..VHVItCBCb8Q5CI-49imar
DtJeSxH2uLU0DhqQP5Zjw4",
                 "eyJhbGciOiJFUzI1NiJ9..ENP0j0-QPsA7N_Mg1-RMN
9IxapeTWtQwR7sPUqEiSNHPuV_fqSdRqqkLOlBdV01cc4lSJdn1XCv-ZHYdZ9t3kA"]
}
```

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

Appendix C. Test Vector Using the ES256 Algorithm

This appendix shows how a signed version of the JSON example object in Section 3.1.1 would look like if applying the ES256 JOSE algorithm [RFC7518] (with a line break in the "signature" property for display purposes only):

```
{
  "statement": "Hello signed world!",
  "otherProperties": [2000, true],
  "signature": "eyJhbGciOiJFUzI1NiJ9..ENP0j0-QPsA7N_Mg1-RMN
9IxapeTWtQwR7sPUqEiSNHPuV_fqSdRqqkLOlBdV01cc4lSJdn1XCv-ZHYdZ9t3kA"
}
```

The example above depends on a JWS header holding the algorithm {"alg":"ES256"}, and the following private key, here expressed in the JWK [RFC7517] format:


```
{
  "kty": "EC",
  "crv": "P-256",
  "x": "6BKxpty8cI-exDzCkh-goU6dXq3MbcY0cd1LaAxiNrU",
  "y": "mCbcvUzm44j3Lt2b5BPyQloQ91tf2D2V-gzeUxWaUdg",
  "d": "6XxMFXhcYT5QN9w5TIg2aSKsbcj-pj4BnZkK7ZOt4B8"
}
```


Note that signing with the ES256 algorithm returns different results for each signature due to a randomization step in the signature computation process.

Appendix D. Enhanced JWS Processing Option

By default, JWS/CT uses the JWS compact serialization mode "as is". As a consequence, a technically redundant, internal-only, base64url encoding step is performed over the JWS Payload. Although the performance hit should be marginal for most real-world applications, a possibility is using the "Unencoded Payload" mode of RFC7797 [RFC7797]. However, this requires that the JWS implementation supports the "b64":false and "crit":["b64"] header elements implied by RFC7797, effectively rendering the RFC7797 mode as an implementer option for specific communities.

Acknowledgements

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

Document History

[[This section to be removed by the RFC Editor before publication as an RFC]]

Version 00:

- Initial publication.

Version 01:

- Added paragraph to Abstract.

*Updated Security Considerations.

Version 02:

*Changed alternative test key to ES256/P-256.

*Moved RFC7797 to an appendix.

*Changed <tt> to only be used on keywords.

*Added some clarity to detached signatures.

Version 03:

*Language changes suggested by ISE.

Version 04:

*Language nit.

Version 05:

*Document refresh.

Version 06:

*Changes after ISE review.

Version 07:

*Changes after ISE and external reviews.

Version 08:

*Changes after ISE and external reviews.

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/