Abstract

The SHREQ specification describes how the JSON Web Signature (JWS) specification combined with the JSON Canonicalization Scheme (JCS), can be utilized to support HTTP based applications needing digitally signed requests. SHREQ is specifically tailored for Web applications using JSON as data interchange format. There is also a SHREQ scheme for HTTP requests that do not have a body ("payload") like GET. SHREQ was designed to be agnostic with respect to REST concepts versus traditional GET/POST schemes.

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 March 11, 2020.

Copyright Notice

Copyright (c) 2019 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust’s Legal Provisions Relating to IETF Documents (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. Code Components extracted from this document must
include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction ........................................... 3
2. Terminology ........................................... 4
3. HTTP Processing ......................................... 4
   3.1. Determining Request Type .......................... 4
   3.2. Return Codes ..................................... 5
4. Processing of JSON Based Requests .......................... 5
   4.1. Request Creation ................................ 6
   4.2. Request Validation ................................ 8
5. Processing of URI Based Requests .......................... 9
   5.1. Request Creation ................................ 10
   5.2. Request Validation ................................ 11
6. Common Operations ....................................... 13
   6.1. Create Signable JSON Data ........................ 13
   6.2. Create Signable URI Data .......................... 13
   6.3. Create HTTP Header Object ......................... 14
   6.4. Create JWS Protected Header ....................... 15
   6.5. Create JWS String ................................ 15
   6.6. Decode JWS String ................................ 16
   6.7. Normalize Target URI .............................. 16
   6.8. Normalize Header Data ............................. 17
   6.9. Validate HTTP Header Object ..................... 18
   6.10. Validate JWS Signature ......................... 19
   6.11. Time Stamps .................................... 20
7. Local Naming Conventions .................................. 21
8. Attachments ............................................... 21
9. IANA Considerations ...................................... 21
10. Security Considerations .................................. 22
11. Acknowledgements ....................................... 22
12. References .............................................. 22
   12.1. Normative References ............................ 22
   12.2. Informal References ............................. 24
Appendix A. Test Vectors ..................................... 24
   A.1. Type=URI, Method=GET, Algorithm=HS256............... 24
   A.2. Type=JSON, Method=POST, Algorithm=ES256 .......... 25
   A.3. Type=JSON, Method=PUT, Algorithm=ES256 ........... 25
   A.4. Type=URI, Method=DELETE, Algorithm=RS256 ......... 26
Appendix B. Other Signed HTTP Request Solutions .................. 27
   B.1. Amazon Web Services .............................. 27
   B.2. HTTP Signatures ................................... 27
   B.3. Open Banking (UK) .................................. 28
   B.4. Financial API ..................................... 28
1. Introduction

Currently there is no standard for digitally signing HTTP [RFC7230] [RFC7231] requests. This has led to the development of a multitude of more or less proprietary solutions (see Appendix B), typically building on using HTTP header data for holding security constructs, while JSON request data is provided in clear in the HTTP body.

SHREQ is intended to provide a standardized alternative, including supporting the REST [REST] concept.

SHREQ builds on a common security model where all elements of an HTTP request are signed:

- HTTP URI.
- HTTP method.
- HTTP body (if applicable).
- Optional: Additional HTTP headers as defined by applications implementing this specification.

In addition there is a mandatory time stamp.

One of the design goals was turning signed requests into self-contained objects. To achieve this for HTTP requests having a JSON [RFC8259] body (see Section 4), the request data also carries the signature. This arrangement has certain implications:

- Signed requests may be stored in databases or be embedded in other JSON objects. The latter includes supporting counter signatures. The canonicalization offered by JCS [JCS] enables validating the integrity of request data at any time.

- For general interoperability concerns as well as due to the reliance on JCS, JSON request data is limited to the I-JSON [RFC7493] subset.

For HTTP requests that do not have a JSON body (see Section 5), the signature and additional request data is added to the original URI [RFC3986], making signed URI-only requests self-contained and serializable as well. For simplicity such requests are (in this specification NB), referred to as URI based requests.
Both variants utilize JWS [RFC7515] for holding the signature data.

For supporting signed HTTP responses any solution may be used. For maximum "symmetry" and code reuse, the [JWSJCS] scheme should be a suitable candidate since it builds on the same building blocks as SHREQ.

The intended audiences of this document are Web tool vendors, as well as designers of secure Web applications.

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. HTTP Processing

The following subsections describe HTTP specifics associated with this specification.

3.1. Determining Request Type

In this specification the distinction between HTTP requests having a JSON body or not is based on the presence of a "Content-Length" header. Requests without a body object are in this specification referred to as URI based requests.

This also implies that not all header combinations permitted by HTTP can be used with this specification:

"Content-Length"
MUST NOT be used with URI based requests. MUST be present for requests having a body and have an argument holding the length of the body in bytes.

"Content-Type"
MUST NOT be used with URI based requests. MUST be present for requests having a body and have the argument "application/json".

"Content-Encoding"
MUST NOT be used with any requests targeting this specification.

"Transfer-Encoding"
MUST NOT be used with any requests targeting this specification.
3.2. Return Codes

This specification utilizes a single HTTP return code 400 (Bad request) for indicating syntax or security errors. Since the number of possible error conditions is significant, it is RECOMMENDED to accompany the error code with a short explanation in "text/plain" format in the HTTP Body like:

- Missing ".secinf" element
- Invalid "alg": es256
- Unknown "kid": example.com:rsa:2018.1
- com.example.jose.Core.validate(2653): Signature validation error
- Missing header variable "x-testing"

Communities using this specification MAY customize error codes if needed. However, in practice, it usually turns out to be of little value compared to a text message and a generic "hard error" code since neither users nor machines can do very much on their own to fix errors that are outside of normal processing.

Application level errors are dealt with in an application specific manner. As an example a bank application which finds out that the customer do not have enough funds to perform a transaction would presumably not return an HTTP error code but rather a specifically crafted error message to be displayed to the user.

Return codes for successful operation are application specific but are typically 200 (OK) or 201 (Created).

4. Processing of JSON Based Requests

Assume there is an unsigned HTTP request like the following:

```plaintext
POST /foo HTTP/1.1
Host: example.com
Content-Type: application/json
Content-Length: 1234

{  
  "something": "data",

      Additional application specific properties

  }

```

Adding a signature to the request above would require the following enhancements to the JSON payload:
{  
"something": "data",

    Additional application specific properties

    "secinf": {  
    "uri": "https://example.com/foo",
    "mtd": "POST",
    "iat": 1551709923,
    "jws": "eyJhbGciOiJIUzI1NiJ9..VHVItCBCb849imarDtjw4"
    }
}

Notes:

  o This specification presumes that request data featured in an HTTP body is expressed as a JSON Object.

  o The "uri" property holds a normalized target URI.

  o The "mtd" property holds the expected HTTP method. In the example it is actually redundant since the absence of a "mtd" property defaults to "POST" for JSON based requests in this specification.

  o The "iat" property holds a time stamp in UNIX "epoch" format.

  o The argument to "jws" (a detached compact JWS) was truncated for brevity.

The following subsections detail the operation for requests having an HTTP body.

4.1. Request Creation

Precondition: the application data to be submitted with the request already exists in a format serializable as a JSON Object, from now on referred to as "message".

In order to create a valid signed JSON request, the following steps (and ordering) MUST be adhered to:

1. Store the target HTTP method in a variable "targetMethod".

2. Store the target URI in a variable "targetURI" after having normalized it as described in Section 6.7.

3. Add a JSON Object called ".secinf" to "message".
4. Add a property "uri" to ".secinf" where the argument is a JSON String holding a copy of "targetURI".

5. Add a property "mtd" to ".secinf" where the argument is a JSON String holding a copy of "targetMethod".

Note: for the HTTP method "POST", this step is optional because "POST" is the default for JSON based requests.

6. If there is a need to include additional HTTP headers in the signed request data, perform the following steps:

* Derive (or define) the hash algorithm to use as described in Section 6.12. Save the algorithm in a variable "hashAlgorithm".

* Create a header object as described in Section 6.3.

7. Add a time stamp property (see Section 6.11) to ".secinf".

Note: if the application data already contains a suitable time stamp property, this step MAY be excluded.

8. Create a "JWS Payload" [RFC7515] as described in Section 6.1.

9. Create a "JWS Protected Header" [RFC7515] as described in Section 6.4.

10. Create a JWS string object as described in Section 6.5.

11. Add a property "jws" to ".secinf" where the argument is a JSON String holding the result of the preceding step.

12. Serialize "message" into an UTF-8 [UNICODE] encoded byte array called "requestData".

13. Submit an HTTP compliant request to the "targetURI" including the following information:

* HTTP method set to "targetMethod".

* HTTP Header "Content-Length" with the argument set to the length of "requestData".

* HTTP Header "Content-Type" with the argument set to "application/json".

* All HTTP headers (if any) specified in step 6.
* Other HTTP headers (if any) needed by the application.

* An HTTP Body containing a copy of "requestData".

4.2. Request Validation

In order to validate a request the following steps MUST be performed:

1. Store the HTTP method of the request in a variable "targetMethod".

2. Store the from the request recreated target URI in a variable "targetURI" after having normalized it as described in Section 6.7.

3. If the HTTP header "Content-Type" is missing or differs from "application/json" the service MUST reject the request (see Section 3.2).

4. If the HTTP header "Content-Length" is missing or malformed the service MUST reject the request (see Section 3.2). Save the length data.

5. Read HTTP body data into a byte array with the length retrieved in the preceding step.

6. Parse the byte array created in the preceding step with a JSON parser and return the result in an object from now on referred to as "message". If there are parsing errors or if "message" is not a JSON Object the service MUST reject the request (see Section 3.2).

7. Using "message", retrieve a JSON Object called ".secinf". If ".secinf" is missing or is not a JSON Object the service MUST reject the request (see Section 3.2).

8. Using ".secinf", the property "jws" is read. If "jws" is missing or is not a JSON String the service MUST reject the request (see Section 3.2). Decode the read string as described in Section 6.6.

9. Using ".secinf", the property "uri" is read. If "uri" is missing or does not match "targetURI" or is not a JSON String the service MUST reject the request (see Section 3.2).

Note: in some proxy arrangements it may be difficult retrieving the proper value of "targetURI". In such cases the comparison with "uri" MAY be disabled.
10. Using "secinf", the property "mtd" is read. If "mtd" is missing the HTTP method is assumed to be "POST" else it is assumed to be the read value. If the derived method does not match "targetMethod" or is not a JSON String the service MUST reject the request (see Section 3.2).

11. If the optional "hdr" property is present in "secinf" perform the following steps:

* Derive the hash algorithm to use as described in Section 6.12. Save the algorithm in a variable "hashAlgorithm".
* Process the argument of "hdr" as described in Section 6.9.

12. Using "secinf", the property "iat" (see Section 6.11) is read. If "iat" is missing or is not a JSON Number the service MUST reject the request (see Section 3.2).

   Note: if the application data already contains a suitable time stamp property, this step MAY be excluded.

13. Remove the "jws" property from "secinf".


15. Perform signature validation as described in Section 6.10.

   Note: validation of application specific data can be performed anytime after step 6. The action(s) to perform after a possible failure is out of scope for this specification (see Section 3.2).

5. Processing of URI Based Requests

Assume there is an unsigned HTTP request like the following:

GET /users?id=435 HTTP/1.1
Host: example.com

The full URI would be as follows:

https://example.com/users?id=435

Adding a signature to this request according to this specification would return the following URI:

https://example.com/users?id=435&.jws=eyJhhiJ.eyJ7fgw.VHVIt
Notes:

- The revised URI represents a complete serializable signed request object.
- The argument to ".jws" (a standard compact JWS) was truncated for brevity.

The middle component of the JWS string ("JWS Payload"), contains Base64Url encoded signed data related to the request. It should (after Base64Url decoding) yield a JSON Object like the following:

```json
{
    "htu": "WUjqfXPztLzzXRCs6EcWCw-GC9hSL7hwCR1nG2FSvQ8",
    "mtd": "GET",
    "iat": 1551863696
}
```

Notes:

- The property "htu" holds a Base64Url encoded value of the normalized target URI after it has been hashed by the hash algorithm associated with the JWS signature.
- The "mtd" property holds the expected HTTP method. In the example it is actually redundant since the absence of a "mtd" property defaults to "GET" for URI based requests in this specification.
- The "iat" property holds a time stamp in UNIX "epoch" format.

The following subsections detail the operation for requests using an HTTP query string component for holding a signature.

### 5.1. Request Creation

In order to create a valid signed URI request, the following steps (and ordering) MUST be adhered to:

1. Store the target HTTP method in a variable "targetMethod".
2. Store the target URI in a variable "targetURI" after having normalized it as described in Section 6.7.
3. Create an empty JSON Object from now on referred to as ".secinf".
4. Derive (or define) the hash algorithm to use as described in Section 6.12. Save the algorithm in a variable "hashAlgorithm".
5. Add a property "htu" (Hashed Target URI) to ".secinf" where the argument is a JSON String holding the outcome of the process described in Section 6.2.

6. Add a property "mtd" to ".secinf" where the argument is a JSON String holding a copy of "targetMethod".

Note: for the HTTP method "GET", this step is optional because "GET" is the default for URI based requests.

7. If there is a need to include additional HTTP headers in the signed request data, create a header object as described in Section 6.3.

8. Add a time stamp property (see Section 6.11) to ".secinf".

9. Serialize the ".secinf" JSON Object into a UTF-8 [UNICODE] byte array representing a "JWS Payload" [RFC7515].

10. Create a "JWS Protected Header" [RFC7515] as described in Section 6.4.

11. Create a JWS string object as described in Section 6.5.

12. Create a query string component by concatenating ".jws=" with the JWS string created in the preceding step. This component is appended to the original unsigned request URI prepended by & or ? depending on if it is the only query component or not.

13. Submit an HTTP compliant request to the target URI including the following information:

* HTTP method set to "targetMethod".
* All HTTP headers (if any) specified in step 7.
* Other HTTP headers (if any) needed by the application.

5.2. Request Validation

In addition to normal validation of received data (which may be carried out before or after the steps outlined here), the following steps MUST be performed in order to validate a URI based HTTP request:

1. Store the HTTP method of the request in a variable "targetMethod".
2. Store the from the request recreated target URI in a variable "targetURI" after having normalized it as described in Section 6.7.

3. Extract the JWS string from the ".jws" element which MUST reside in the query string of "targetURI". If the JWS string is missing the service MUST reject the request (see Section 3.2).

4. Decode the argument of the preceding step as described in Section 6.6.

5. Remove the ".jws" query string component from "targetURI". Note that if the ".jws" query component is the last part of "targetURI", the delimiter immediately preceding the ".jws" component is removed, else the succeeding delimiter is removed.

6. Parse "JWS Payload" (created in step 4) with a JSON parser and from now on refer to the result as ".secinf". If ".secinf" is not a JSON Object the service MUST reject the request (see Section 3.2).

7. Derive the hash algorithm to use as described in Section 6.12. Save the algorithm in a variable "hashAlgorithm".

8. Using ".secinf" the property "htu" is read. If "htu" is missing or is not a JSON String the service MUST reject the request (see Section 3.2).

9. Perform the operation described in Section 6.2 and compare the outcome with the argument to "htu". If these value do not match the service MUST reject the request (see Section 3.2).

   Note: in some proxy arrangements it may be difficult retrieving the proper value of "targetURI". In such cases the comparison with "htu" MAY be disabled.

10. Using ".secinf", the property "mtd" is read. If "mtd" is missing the HTTP method is assumed to be "GET" else it is assumed to be the read value. If the derived method does not match "targetMethod" or is not a JSON String the service MUST reject the request (see Section 3.2).

11. If the optional "hdr" property is present in ".secinf", process the argument of "hdr" as described in Section 6.9.

12. Using ".secinf", the property "iat" (see Section 6.11) is read. If "iat" is missing or is not a JSON Number the service MUST reject the request (see Section 3.2).
13. Perform signature validation as described in Section 6.10.

Note: validation of application specific data can be performed anytime. The action(s) to perform after a possible failure is out of scope for this specification (see Section 3.2).

6. Common Operations

This specification builds on a modular scheme using common procedures described in the following subsections.

6.1. Create Signable JSON Data

Unsigned request data is now supposed to reside in "message". To facilitate resilience against (legitimate) variances in JSON processing between different platforms and systems, "message" needs to be canonicalized and serialized into a UTF-8 [UNICODE] encoded byte array. If the used JSON tools offer intrinsic support for JCS [JCS], this is typically a single operation, else the followings steps are performed:

1. Serialize "message" using standard JSON tools for the platform.

2. Create a canonical and UTF-8 encoded form of the data created in the preceding step, through an external software solution supporting JCS.

The output from JCS represents a "JWS Payload" [RFC7515].

6.2. Create Signable URI Data

For URI based requests, the steps to create signable URI data are as follows:

1. Convert "targetURI" into a UTF-8 [UNICODE] encoded byte array.

2. Create a digest of the result of the preceding step using the previously defined "hashAlgorithm". The result is a byte array.

3. The result of the preceding step is subsequently Base64Url encoded.

The test vectors in Appendix A provide a few examples showing authentic values of the "htu" (Hashed Target URI) property.
6.3. Create HTTP Header Object

To create a digest of headers to be included in a signed request, perform the following operations:

1. Create an empty string "headerBlob".
2. Create an empty string "headerList".
3. Create a collection of headers to be sent as described in Section 6.8.
4. Enumerate the "headerCollection" and perform the following steps for each entry:
   * Append header field name to "headerList".
   * Append header field name to "headerBlob".
   * Append a semicolon (':') to to "headerBlob".
   * Append header field value to "headerBlob".
   * For all but the last entry, append a newline (U+000A) to "headerBlob".
   * For all but the last entry, append a comma (',') to "headerList".
5. Create a two element JSON Array object "headerData".
6. Run the previously defined "hashAlgorithm" (see Section 6.12) over the UTF-8 [UNICODE] representation of "headerBlob".
7. Base64Url-encode the result of the preceding operation and assign the result to the first entry in "headerData" in the form of a JSON String.
8. Assign a copy of "headerList" to the second entry in "headerData" in the form of a JSON String.
9. Add a property "hdr" to "secinf" using a copy of "headerData" as argument.

Below is an example of header input data:

```plaintext
x-debug: full
Cache-Control: max-age=60, must-revalidate
```
Applying the process described in this subsection and using the SHA-256 [SHS] hash algorithm should generate the following ".secinf" data:

"hdr": ["Ljzuq8C9FScbvLpBxGBGNOs-WQUd7g17R6dizahhe-0", "x-debug,cache-control"]

6.4. Create JWS Protected Header

Create a "JWS Protected Header" [RFC7515] JSON Object with algorithm and key data adapted for the application. Below is a minimal example:

```json
{
    "alg": "ES256"
}
```

6.5. Create JWS String

To create a compact JWS object (a string), perform the following steps:

1. Serialize the previously defined "JWS Protected Header" object into a UTF-8 [UNICODE] encoded byte array.

2. Base64Url-encode the output from the preceding step into a local variable "jwsProtectedHeaderB64U".

3. Base64Url-encode the previously defined variable "JWS Payload" into a local variable "jwsPayloadB64U".

4. Set a local variable "signedData" to the UTF-8 encoded representation of the concatenation of:
   * The previously defined variable "jwsProtectedHeaderB64U".
   * A point character (".").
   * The previously defined variable "jwsPayloadB64U".

5. Use the designated signature key, signature algorithm and "signedData" to create a "JWS Signature" object (byte array).

6. Return the string consisting of the concatenation of:
   * The previously defined variable "jwsProtectedHeaderB64U".
   * A period character ('.').
* For URI based requests only: "jwsPayloadB64U". That is, JSON based requests use the detached JWS format described in Appendix F of [RFC7515].

* A period character (‘.’).

* The previously defined variable "JWS Signature", here encoded in Base64Url [RFC4648].

6.6. Decode JWS String

The following processing steps presume that there is an input string holding a JWS compact object, here called "jwsString":

1. Verify that "jwsString" has the syntax

   "header.payload.signature"

   where the length of the "payload" element is zero for JSON based requests and non-zero for URI based requests. That is, JSON based requests use the detached JWS format described in Appendix F of [RFC7515].

2. Assign the "header" portion of "jwsString" to a variable "jwsProtectedHeaderB64U".

3. Base64Url-decode "jwsProtectedHeaderB64U" into a byte array.

4. Parse the output from the preceding step with a JSON parser and assume that the result (which MUST be a JSON Object) represents a "JWS Protected Header" [RFC7515].

5. For URI based requests only:
   base64Url-decode the "payload" portion of "jwsString" into a byte array representing a "JWS Payload" [RFC7515].

6. Base64Url-decode the "signature" portion of "jwsString" into a byte array representing a "JWS Signature" [RFC7515].

   If any of the steps above fail, the service MUST reject the request (see Section 3.2).

6.7. Normalize Target URI

To facilitate comparison between actual (received) URIs and signed URIs, URIs MUST be normalized according to the following:
The schema default ports 443 and 80 MUST be removed from HTTPS and HTTP URIs respectively.

URI characters that have been escaped that are in the non-reserved set [ALPHA DIGIT ‘-’ ‘.’ ‘_’ ‘~’] MUST be restored in their natural form.

Escape sequences MUST transformed into uppercase.

Non-ASCII characters MUST be escaped to their UTF-8 [UNICODE] counterpart.

Host names MUST be lowercased.

The following URI shows a non-normalized URI:

https://EXAMPLE.COM:443/%63EURO%2f

Note: EURO denotes a single Euro character (Unicode: U+20AC), which not being ASCII, is currently not displayable in RFCs.

The same URI after normalization:

https://example.com/c%E2%82%AC%2F

[[ This section is still incomplete ]]

6.8. Normalize Header Data

Headers to be included in signed requests MUST be normalized. This subsection shows a common procedure for senders and receivers based on Section 3.2.4 of [RFC7230].

Collect received headers or headers to be submitted in a list of header field name and header field value pairs according to the following:

- Header field names MUST be lowercased.

- Leading and trailing optional whitespace (OWS) in the header field value MUST be omitted. If there are multiple instances of the same header field name, all header field values associated with the header field name MUST be concatenated, separated by an ASCII comma and an ASCII space (’, ’), and used in the order in which they are intended to appear in an HTTP message. Any other modification to the header field value MUST NOT be made.
This list is referred to as "headerCollection" in other places in this specification.

Below is an example of header input data:

x-debug: full
Cache-control: max-age=60
Cache-Control: must-revalidate

Applying the process described in this subsection should generate the following collection:

<table>
<thead>
<tr>
<th>Header Field Name</th>
<th>Header Field Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>x-debug</td>
<td>full</td>
</tr>
<tr>
<td>cache-control</td>
<td>max-age=60, must-revalidate</td>
</tr>
</tbody>
</table>

For interoperability reasons it is RECOMMENDED to not use duplicate header names for headers that are to be signed. Apparently proxy servers do not always honor original header ordering.

### 6.9. Validate HTTP Header Object

To validate a digest of headers in a signed request, perform the following operations:

1. Create a collection of received headers as described in Section 6.8.

2. Create an empty string "headerBlob".

3. Read the "hdr" property of ".secinf". This MUST be a JSON Array holding exactly two JSON String elements.

4. Perform the following actions on the data obtained in the preceding step:

   * Base64Url-decode the first string into a byte array "digest".
   * Split the second string into ordered array of strings called "headerList". Note that the format MUST be a list of header field names in lowercase, separated by comma (",") characters. There MUST NOT be any whitespace or terminating comma in this string.
* Verify that the received "headerList" contains the header field names as defined by an application specific policy.

5. Enumerate the "headerList" and perform the following steps for each entry:

* Append header field name to "headerBlob".
* Append a semicolon (':') to to "headerBlob".
* Retrieve the matching header field value from "headerCollection".
* Append the result of the preceding step to "headerBlob".
* For all but the last entry, append a newline (U+000A) to "headerBlob".

6. Run the previously defined "hashAlgorithm" (see Section 6.12) over the UTF-8 [UNICODE] representation of "headerBlob".

7. Verify that the result of the preceding step is identical to "digest".

If any of the steps above fail, the service MUST reject the request (see Section 3.2).

Note that this specification does not enforce any particular ordering of signed header elements.

6.10. Validate JWS Signature

Validation of the JWS [RFC7515] object, using the previously extracted and decoded objects requires the following steps:

1. Verify that the received "JWS Protected Header" contains a JWS algorithm ("alg") and key identifiers that matches the needs of the application.

2. Retrieve the signature validation key. This part is application specific since the key may be implicit, specified by a key ID ("kid") or be supplied in a certificate path ("x5c").

3. Set a local variable "signedData" to the UTF-8 [UNICODE] encoded representation of the string created by concatenating the following elements:

* The previously collected variable "jwsProtectedHeaderB64U".
4. Validate the signature using the algorithm retrieved in step 1, the signature validation key from step 2, "signedData" from step 3 and the previously collected "JWS Signature" object (byte array).

If any of the steps above fail, the service MUST reject the request (see Section 3.2).

6.11. Time Stamps

Time stamps have the same name ("iat"), format and function as described in JWT [RFC7519], Section 4.1.6. However, in this specification time stamps are REQUIRED, and stored in the ".secinf" JSON Object.

Although JWT permits non-integer values, implementers of this specification SHOULD limit generated time stamp granularity to seconds and use integer representation.

The policy with respect to the difference between the current time and received time stamps is out of scope for this specification. However, for security reasons it is generally a good idea limiting deviations to a few minutes as well as using network based clock synchronization in both ends.


Inclusion of HTTP header elements as well as the "htu" property of URI based requests depends on digests produced by a hash algorithm. The default is using the hash algorithm associated with the JWS signature algorithm ("alg") featured in the "JWS Protected Header". That is, the JWA [RFC7518] algorithms "ES256" and "HS384" imply the hash algorithms SHA-256 and SHA-384 respectively.

In case this is not desired, this specification permits overriding the default by including a "hao" (Hash Algorithm Override) property in the ".secinf" JSON Object. The currently recognized arguments to "hao" are:

"S256" for SHA-256 [SHS]

"S384" for SHA-384 [SHS]
"S512" for SHA-512 [SHS]

7. Local Naming Conventions

Although using the ".secinf" JSON property name and ".jws" query component name is RECOMMENDED, this specification permits (=being considered as compatible), the use of local naming conventions as long as the specified procedures and formats are adhered to.

Local naming conventions MUST be properly communicated in the community using them.

8. Attachments

[[]

It may be possible to extending the JSON based request to also support attachments using MIME multipart schemes. This is though currently out of scope for this specification.

An alternative to attachments is featuring such data in Base64Url encoded fields.

Recently, "cloud" based schemes using (preferably time-limited) URLs with hard-to-guess nonce values have become a viable method for supporting related additional data. By combining hash values with such URLs, integrity of the additional data can be verified.

]]

9. IANA Considerations

This document currently has no IANA actions but the reserved names below could be candidates for IANA registration:

.secinf
  JSON Object holding the security related data of this specification.

.jws
  HTTP query component holding the security related data of this specification.

The hash algorithms defined in Section 6.12 could also benefit from IANA registration.
10. Security Considerations

The purpose of this specification is adding an integrity and authorization layer to HTTP requests. This part is subject to the same security considerations as the underpinning JCS and JWS schemes.

For most applications HTTPS [RFC7231] would be the logical choice, not only for protecting application data from snooping, but also to not unnecessarily reveal data about signature keys.

In a cloud scenario with Web servers open for access by any party new security challenges are introduced. Cryptographic solutions protect data but may also add vulnerabilities to denial-of-service attacks since they often need substantial processing.

Protecting against replay attacks is important because replay may actually be a legitimate facility for systems repeating a request due to a communication failure. This cannot be entirely solved by time stamps and cryptography; you usually need unique transactions IDs and database support as well. For reliable operation there must be common rules within a community using such features. The REST [REST] paradigm also requires such measures due to the idempotent operation specified for "PUT", "GET" and "DELETE.

11. Acknowledgements

Parts of this specification were derived from the HTTP signature [HTTPSIG] draft.

12. References

12.1. Normative References


12.2. Informal References


12.3. URIs


Appendix A. Test Vectors

The following test vectors "activate" all parts of the specification. After removing the line breaks needed for publishing, the test vectors are supposed to be fully validatable.

A.1. Type=URI, Method=GET, Algorithm=HS256

Target URI:

https://example.com/users/456

Signed URI:

https://example.com/users/456?.jws=eyJhbGciOiJIUzI1NiJ9.X3x.bXpkldk7NswxGK7uJ89J8Ew==
Decoded JWS Payload:

```
{
  "htu": "fiVi4jYhDt7VCuQIKUIDWINEWfOh_NXHfLTZNEeSavY",
  "iat": 1551951900
}
```

Symmetric signature validation key, here in hexadecimal notation:

```
7fdd851a3b9d2dafc5f0dc0030e22b9343900cd4ede4948568a4a2ee655291a
```

A.2. Type=JSON, Method=POST, Algorithm=ES256

Target URI:

```
https://example.com/users
```

JSON Body:

```
{
  "name": "John Doe",
  "profession": "Unknown",
  "secinf": {
    "uri": "https://example.com/users",
    "iat": 1551951900,
    "jws": "eyJhbGciOiJFUzI1NiJ9..-N7yuF1TEASo5Ub5q2T1_EkLWrWHs2
nyHjDwpkinoRcQbSo8h2ygL9pmGzd_YU4jn_bcMQF8BrTI1SioNe15GQ"
  }
}
```

Public signature validation key, here in PEM format:

```
-----BEGIN PUBLIC KEY-----
MFkwEwYHKoZIzj0CAQYIKoZIzj0DAQcDQgAEcensDzcMEkgiePz6DXB7cDuwFems
hAFR90UNVQFCg8TGrv9N77uT55xIXvYnvuAqIPQgefOnAdpTu3qdV5g==
-----END PUBLIC KEY-----
```

A.3. Type=JSON, Method=PUT, Algorithm=ES256

Target URI:

```
https://example.com/users/456
```

JSON Body:
{  
"name": "Jane Smith",
"profession": "Hacker",
"secinf": {  
"uri": "https://example.com/users/456",
"mtd": "PUT",
"iat": 1551951900,
"jws": "eyJhbGciOiJFUzI1NiJ9.._VWTXYcgr60TCcJg6XZzPkHsLU-jUTT1HoQ92bihMIDlXR7wNfmxlHWSUc9cyFCxzsBy9yq33eFn3fApIH42SA"  
}  
}  

Public signature validation key, here in PEM format:

-----BEGIN PUBLIC KEY-----  
MFkwEwYHKoZIzj0CAQYIKoZIzj0DAQcDQgAEcensDzcMEkgiePz6DXB7cDuwFems hAFR90UNVQFCg8TGrtyN7p7AbT55VxIXvYnuAQIPQef0AdpTu3qdV5g==  
-----END PUBLIC KEY-----  

A.4. Type=URI, Method=DELETE, Algorithm=RS256  

Target URI:  

https://example.com/users/456  

Signed URI:  

https://example.com/users/456?.jws=eyJhbGciOiJFUzI1NiJ9.eyJodHUiOiI5R3FtRDBSRWRqSDFZNklvSXROjKURuU0pjVzNuShnoM08eHQ3Zk1RQ1cy N3FtOEQyWUNtNh4RzRwU1hwoGJFM3lTT3RzWlhIR0VJSWw1M05jQSIsIm10ZCI6 IkRFTEVURSIsmIhdCI6MTU1MTkiMTkwMCwiaGFvIjoiuXMiIsImhkiI6WIyIz ZXBrqno4RUjwMUxXY01EdFdi1WnFW2jFLYjJya1FNZzE5RjVvT2Fhbk91SVFps1Zl SHBrSG5wWzLMZzB2zBwOG5xSx2ZEx2EfXvVQxNFU5Vxg5USIsIngZGVidWci XXO.YRTEbCrOy11c0HcPSDk_Dct1565qmcYWFcuGwqsgg7vnC2rv2vdCehE1xjLUXcMrhp_gSmdkyyJQ06xetax-nMmXUhrDtcRoeCfo12-xDymZTJxtd1lx6Pnmt9CnM4Z0VrVjro7eI0WhCc4p5As7zDS2arNM-IsWiNJ1T25Edb8ZS7kLbSA6lIm l03io885kCO0HN1q-1zeaOX3DnlltMJKZQzrlTlXv2o0qJ3kL8bxF6ag10FCO z0YUU2kici55jvcBgwuepcFw-rN5QEg8PzA8YA-nGFWHBpxJUWWeYqXzudRcQQZtms7Yc1yK7z3fNhht60h1A  

Decoded JWS Payload:
Note the overridden hash algorithm.

Required HTTP Headers:

x-debug: full

Public signature validation key, here in PEM format:

```
-----BEGIN PUBLIC KEY-----
MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAhFWEXArvaZEpSP5qNX7x
4C4H128GJQTNvnDwkJfqiW63kXbdyreS06bz6GnY3tfQ/093GauWimgKBmGAGM
PtsV83Qxwl01eO4ujiB9pema0qTvs0MW1Hxk12GFkYfAmbu0EufYDLeDHe0bk
kXbS1b7/t8pC5v8c8HLgHjEgqy0I1FwjrR0D+uLo+xgSbpmCtYkB5icT/zFgpRgY4
zJNLIv7GzlZ2S4C5ArGj341lL47+L8bozuYjqNOv9sqX0Zgli15XajJndvrr7UqZu
1xQFMg38reoM3iarBP/5kEFlb/v9iak602VO3k28fQhMaocP7JWR2YLZ3n0+WT
FwIDAQAB
-----END PUBLIC KEY-----
```

**Appendix B.  Other Signed HTTP Request Solutions**

This appendix briefly outlines a few other solutions addressing Signed HTTP Requests.

**B.1. Amazon Web Services**

AWS provides a system for their clients using HTTP headers holding security constructs while the digested HTTP body may hold any valid media type. For more information see the [AWS]. Signatures may also be added to query strings in a similar fashion to Section 5.

**B.2. HTTP Signatures**

HTTP Signatures is a system using HTTP headers holding security constructs while the (optional) digested HTTP body may hold any valid media type. This scheme has been adopted by the French Open Banking API [STET]. For more information see the Internet draft [HTTPSIG]. HTTP Signatures also supports signed response data.
B.3. Open Banking (UK)

The current (3.1) version of the Open Banking API [OBIE] use a scheme where a dedicated HTTP header holds a detached JWS signature covering a clear text JSON message in the HTTP body:

```
POST /foo HTTP/1.1
Host: example.com
Content-Type: application/json
x-jws-signature: eyJhbGciOiJSUzI1N...SD7xMbpL-2QgwUsA1MGzw
Content-Length: 2765

{
  "something": "data",

  Additional application specific properties
}
```

Notes:
- The HTTP URI, method and headers are unsigned.
- The signature argument (a JWS) was truncated for brevity.

B.4. Financial API

The current version (Draft 06) of the financial API [FAPI] use a scheme where the payload is signed using JWS in Base64Url mode:

```
POST /foo HTTP/1.1
Host: example.com
Content-Type: application/jws
Content-Length: 1288

eyJhbGcRjIn0.ew0KICJfds56gty5ypc3MiOiA.2QgwUsA5656561MGzw
```

Notes:
- The HTTP URI, method and headers are unsigned.
- The JWS signature was truncated for brevity.

Appendix C. Development Portal

The SHREQ specification is currently developed at: https://github.com/cyberphone/ietf-signed-http-requests [1].
Author’s Address

Anders Rundgren
Independent
Montpellier
France

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