LD Signature Format Alignment

An Abstract from Rebooting the Web of Trust IV Design Workshop

By Kim Hamilton Duffy, Rodolphe Marques, Markus Sabadello, Manu Sporny

ABSTRACT

The goal of the "LD Signature Format Alignment" Working Group at Rebooting the Web of Trust IV was to investigate the feasibility and impact of the proposed 2017 RSA Signature Suite spec, which brings JSON-LD signatures into alignment with the JOSE JSON Web Signature (JWS) standards.

The 2017 RSA Signature Suite is based on RFC 7797, the JSON Web Signature (JWS) Unencoded Payload Option specification.

This approach avoids past concerns about JWT raised in the LD signature adopters, including:

- Increased space consumption associated with base-64 encoding.
- Difficulty of nesting or chaining signatures, leading to data duplication.
- Use of a format that is not a JSON object, preventing ability to rely exclusively on a JSON document-based storage engine (while preserving the signature).

Using unencoded payloads with detached content, as described in the introduction of RFC 7797, addresses these concerns and helps in cases in which "...the payload may be very large and where means are already in place to enable the payload to be communicated between the parties without modifications." This avoids unnecessary copying and transformations which can result "significant space and time improvements" when working with large payloads.

Our working group had two primary questions about the proposed 2017 RSA Signature Suite:

1. Is the specification sufficiently clear for implementors?
2. Is there a negative usability impact to LD signature implementations using this signature suite?

To answer these questions, we developed prototypes for the suite in several key programming languages to assess:

- Availability of library support for JWS unencoded payload options
- Impact to existing LD signature implementations, e.g. jsonld-signatures library
- Impact to usability of Verifiable Claims (and others) using JSON-LD signatures with this signature suite.

STATUS

We accomplished our goals as follows:

1. We delivered prototypes for the 2017 RSA Signature Suite that provided sufficient confidence to move forward with the proposed aligned signature approach.
2. We verified that there was no significant impact to existing LD signature implementations and usability in general. Specifically, unencoded payloads with detached content allows LD signatures to be compatible with JWS while avoiding the concerns raised in the summary of past concerns described above.

The major obstacle we encountered while performing this work was the lack of JSON Web Signature library support for unencoded payloads, which is addressed in "Next Steps".

Implementations of LD JWS signing

The following prototypes were developed:

- For Javascript/Node.js: https://github.com/WebOfTrustInfo/ld-signatures-js (this is a fork of JSON-LD signatures official library)
- For Python: https://github.com/WebOfTrustInfo/ld-signatures-python
- For Java: https://github.com/WebOfTrustInfo/ld-signatures-java

JSON-LD JWS Implementation Guidance (cross-platform)

A white paper, which follows, describes the precise differences between existing LD signatures and the new approach.

NEXT STEPS

The primary gap in developing these prototypes, which accounted for most of our development work, was lack of library support for JWS unencoded payloads. To work around this limitation, our implementations mirrored the only implementation we found, available in the JOSE PHP library.

A cleaner solution that we propose is to recraft our prototypes as JWS unencoded payload libraries. Such a library would expose simple sign-and-verify APIs, for example:

```
signature = sign(headers: JSON,
payload: STRING);
```

In this example, payload is assumed to be a detached payload, as described in RFC 7797.

This library would facilitate minimal changes to existing JSON-LD signature implementations.

Detailed List of Next Steps

- Determine how to address problem that JWS implementations lack support for RFC 7797:
- Recraft prototypes as JWS unencoded-signature libraries to provide a RFC 7797 implementation (with a least RS256) to either be merged into official JWS libraries or to act as standalone bridges until official support is provided.
- Double-check end-to-end samples with RS256 algorithm (not provided in RFC 7797 or PHP tests).
- Add 2017 RSA Signature suite to JSON-LD signature libraries, consuming JWS unencoded payload implementation.

Implementing the 2017 RSA Signature Suite in a LD Signature Library

A White Paper from Rebooting the Web of Trust IV Design Workshop

By Kim Hamilton Duffy, Rodolphe Marques, Markus Sabadello

This document describes specific steps and issues with implementing the 2017 RSA Signature Suite in an existing LD signature library.

SOURCE OF TRUTH

RFC 7797 does not include an RS256 example, so we obtained a source of truth using the JOSE library, which is the only library we located implementing the RFC 7797 spec. We used the test testCompactJSONWithUnencodedDetachedPayload from tests/Functional/SignerTest.php, which uses the algorithm HMAC-SHA256, as the basis for a new test using the RSA-SHA256 algorithm.

The resulting unit test is shown in the included testCompactJSONWithUnencodedDetachedPayload RS256.

```
public function testCompactJSONWithUnencodedDetachedPayloadRS256()
{
    $payload = '$.02';
    $protected_header = [
        'alg'  => 'RS256',
        'b64'  => false,
        'crit' => ['b64'],
    ];

    $key = JWKFactory::createFromKeyFile(
        __DIR__.'/../Unit/Keys/RSA/private.encrypted.key',
        'tests', // password for key
        [
            'kid' => 'My Private RSA key',
            'use' => 'sig',
        ] // these options do not affect outcome of this test
    );

    $jws = JWSFactory::createJWSWithDetachedPayloadToCompactJSON($payload, $key,
$protected_header);
$this->assertEquals('eyJhbGciOiJSUzI1NiIsImI2NCI6ZmFsc2UsImNyaXQiOlsiYjY0Il19..f
ZRkjTTrcXdUovHjghM6JvlMhJuR1s8X1F4Uy_F4oMhZ9KtF2Zp78lYSOI7OxB5uoTu8FpQHvy-dz3N4n
LhoSWAi2_HrxZG_2DyctUUB_8pRKYBmIdIgpOlEMjIreOvXyM6A32gR-PdbzoQq14yQbbfxk12jyZSwc
aNu29gXnW_uO7ku1GSV_juWE5E_yIstvEB1GG8ApUGIuzRJDrAAa8KBkHN7Rdfhc8rJMOeSZI0dc_A-Y
7t0M0RtrgvV_FhzM40K1pwr1YUZ5y1N4QV13M5u5qJ_lBK40WtWYL5MbJ58Qqk_-Q8l1dp6OCmoMvwdc
7FmMsPigmyklqo46uyjjw', $jws);

    $loader = new Loader();
    $loaded = $loader->loadAndVerifySignatureUsingKeyAndDetachedPayload(
        $jws,
```

```
            $key,
            ['RS256'],
            $payload,
            $index
        );

        $this->assertInstanceOf(JWSInterface::class, $loaded);
        $this->assertEquals(0, $index);
        $this->assertEquals($protected_header, $loaded->getSignature(0)-
>getProtectedHeaders());
    }
```

Note this test uses the `{"alg":"RS256","b64":false,"crit":["b64"]}` header and `$.02` as the unencoded payload.

The test asserts that the input to the signing function, including the protected headers, should match:

```
eyJhbGciOiJSUzI1NiIsImI2NCI6ZmFsc2UsImNyaXQiOlsiYjY0Il19.$.02
```

And the resulting signature should match:

```
eyJhbGciOiJSUzI1NiIsImI2NCI6ZmFsc2UsImNyaXQiOlsiYjY0Il19..fZRkjTTrcXdUovHjghM6Jv
lMhJuR1s8X1F4Uy_F4oMhZ9KtF2Zp78lYSOI7OxB5uoTu8FpQHvy-dz3N4nLhoSWAi2_HrxZG_2DyctU
UB_8pRKYBmIdIgpOlEMjIreOvXyM6A32gR-PdbzoQq14yQbbfxk12jyZSwcaNu29gXnW_uO7ku1GSV_j
uWE5E_yIstvEB1GG8ApUGIuzRJDrAAa8KBkHN7Rdfhc8rJMOeSZI0dc_A-Y7t0M0RtrgvV_FhzM40K1p
wr1YUZ5y1N4QV13M5u5qJ_lBK40WtWYL5MbJ58Qqk_-Q8l1dp6OCmoMvwdc7FmMsPigmyklqo46uyjjw
```

Our prototypes successfully matched this testcase, and matched results on JSON-LD claim inputs.

LD SIGNATURE FLOW

Overview

The signing flow for the 2017 RSA Signature Suite is identical to other signature suites in the JSON-LD signature library; the processing required to implement 2017 RSA Signatures is confined to step #5 bolded below (all other steps are unchanged). A new algorithm, `RsaSignature2017`, was added to implement this signature suite.

The LD signature algorithm works as follows:

Inputs: - JSON-LD headers (nonce, created, creator, ...) same as before, algorithm should be `RsaSignature2017` - JSON-LD document

JSON-LD Signing Algorithm:

1. Ensure algorithm is in accepted set
2. Add `created` date of now, if not supplied
3. Canonicalize using the `GCA2015` algorithm, as specified in the 2017 RSA Signature Suite specification (NOTE: `GCA2015` was formerly called `URDNA2015`)
4. Prepend the JSON-LD signature options `date`, `domain`, `nonce` to the input to sign, as implemented in the `_getDataToHash` method of the JSON-LD signature library
5. Sign with the 2017 RSA Signature Suite (details in next section)
6. Compact the signature

Outputs: - JSON-LD document with the signature block added

We'll refer to steps 2-4 of the JSON-LD signing algorithm as the "JSON-LD canonicalized form" of the JSON-LD document.

Signing with the 2017 RSA Signature Suite

This section drills into step #5 above.

To extend the JSON-LD signature library to support the 2017 RSA Signature Suite, we added a new algorithm type -- `RsaSignature2017` -- and a new processing case for this type in the function `createSignature`.

First, suppose a JWS library with unencoded payload support were available. If so, then the steps would be:

1. Form the JWS Headers

Per RFC 7797, creating a JWS signature using the unencoded payload option requires the JWS Header parameters `"b64":false` and `"crit":["b64"]`. In addition to these parameters, `RsaSignature2017` specifies using RSA Signatures with SHA-256. This corresponds to a JWS signing algorithm of `RS256`.

In sum the complete set of JWS headers used for a 2017 RSA Signature is:

```
{
    "alg":"RS256",
    "b64":false,
    "crit":["b64"]
}
```

2. Call the JWS library with headers from #1 (parameter 1: `headers`) and the JSON-LD canonicalized payload (parameter 2: `payload`)

```
result = sign(headers: JSON,
payload: STRING);
```

3. Update the LD `signature` block to contain `signatureValue=<result>`

Implementing JWS unencoded payload signing

In step #2 above, we assumed the availability of a JWS library supporting unencoded payloads. Because we only found a PHP library supporting unencoded payloads, we needed to implement those steps in the language we chose.

Per RFC 7797, when the `b64` header parameter is used, it must be integrity protected. Therefore it must occur within the JWS Protected Header (meaning it is part of the input that is signed). Also, per RFC7797, the expected input to sign is formatted as follows:

```
ASCII(BASE64URL(UTF8(JWS Protected
Header)) || '.' || JWS Payload
```

In our case, `JWS Payload` is the JSON-LD canonicalized form.

This yields the following steps:

1. Format the input to sign:
    a) Stringify the JWS headers, sorting the keys.
        - Note: sorting the header parameters is an implementation choice to allow predictability in the sorting order of the protected headers. Since the original JWS header can be obtained from the JWS signature prefix, verification could simply ensure it encodes the JWS headers in the same order.
    b) Encode the stringified header, referred to as <header> below, as follows:
        - utf-8 encode
        - base64 url encode
        - ascii encode
    c) Form the JWS input to sign as `<header> + "." + <payload>`, where <payload> is the JSON-LD canonicalized form.
        - The critical distinction here is that `payload` is not base64 encoded, per the b64=false argument.
2. Sign:
    a) RSASHA256-sign the JWS input
    b) base64-url-encode the signature value
3. Return the signature result `<header> + ".." + <base64Signature>`:
    a) The `..` indicates a JWS detached payload.

Note that typically in JWS, the encoded payload is between the middle 2 dots.

STEPS TO VERIFY

The verification algorithm uses the following steps:

1. Record the 'signatureValue' field from the 'signature' section of the JSON-LD document, then remove the entire 'signature' section. Recall the signature value is the 'base64Signature' portion of the JWS signature, i.e. excluding "<header>.." in: <header> + ".." + <base64Signature>

2. Follow the same steps as in signing listed in "1. Format the input to sign", yielding the JWS input.

3. Using a RSA256 signature library, call its "verify" method with the JWS input and the expected signature from step 1. This returns a boolean indicating whether the signature matches.

PROBLEMS ENCOUNTERED

Lack of JWS detached payload library support

As described above, the only library we found that supports detached payloads was the PHP JOSE library.

Inconsistent ordering of JWS headers

To our knowledge the JOSE specs do not specify how JSON headers should be ordered. In our implementations, we ensured consistent lexicographical sorting of JWS headers. This is not critical since the encoded header is included in the signature, but our goal was to produce consistent signatures (similar to what's done in `_getDataToHash`.

Specifying the sorting of the keys, the separators and the encoding should be enough for any implementation to be able to produce the same signature.

Example in python:

```
import json

header = {'alg': 'RS256', 'b64': False, 'crit': ['b64']}

# stringify json
# there are no guarantees about the ordering of the keys and the separators use
# a whitespace between the keys
json.dumps(header)
'{"crit": ["b64"], "alg": "RS256", "b64": false}'

# we can specify the separators. In this case we say we don't want whitespaces
json.dumps(header, separators=(',', ':'))
'{"crit":["b64"],"alg":"RS256","b64":false}'

# and we can specify the ordering of the keys
json.dumps(header, separators=(',', ':'), sort_keys=True)
'{"alg":"RS256","b64":false,"crit":["b64"]}'

# ultimately we can specify the encoding to use and return a bytestring that
can then be used to base64 encode / sign / hash
json.dumps(header, separators=(',', ':'), sort_keys=True).encode('utf-8')
b'{"alg":"RS256","b64":false,"crit":["b64"]}'
```

REFERENCE: MODIFICATIONS TO JAVASCRIPT JSON-LD SIGNATURE LIBRARY TO SUPPORT 2017 RSA SIGNATURE SUITE

The ld-signatures-js repo contains the 2017 RSA Signature Suite prototype

The modifications are: - Add new algorithm type `RsaSignature2017` - Add new paths to `_createSignature` to support `RsaSignature2017` (Node.js and Javascript environments) - Add new paths to `_verifySignature` to support `RsaSignature2017` (Node.js and Javascript environments)

For example, the inlined implementation of `_createSignature` with algorithm `RsaSignature2017` (Node.js environment) is:

```
var crypto = api.use('crypto');
var signer = crypto.createSign('RSA-SHA256');

// detached signature headers for JWS
var protectedHeader = {"alg":"RS256","b64":false,"crit":["b64"]};

var stringifiedHeader = JSON.stringify(protectedHeader,
Object.keys(protectedHeader).sort());
var b64UrlEncodedHeader = base64url.encode(stringifiedHeader);

// jws input to sign
var to_sign = b64UrlEncodedHeader + "." + _getDataToHash(input, options);

// sign
signer.update(to_sign, 'utf-8');
var signaturePart = signer.sign(options.privateKeyPem, 'base64');

// JWS signature for unencoded payload is: <b64UrlEncodedHeader> + '..' +
<signaturePath>
var signature = b64UrlEncodedHeader + ".." + signaturePart;
```

Reminder: This inlined version is to demonstrate the computations performed. It includes steps that should be performed by a JWS library supporting unencoded payloads. The ld-signatures-js repo factors these parts out as separate functions, but should ultimately be replaced by a proper JWS library supporting unencoded payloads, when a javascript implementation exists.

ADDITIONAL CREDITS

Abstract Authors: Kim Hamilton Duffy, Rodolphe Marques, Markus Sabadello, Manu Sporny

White Paper Authors: Kim Hamilton Duffy, Rodolphe Marques, Markus Sabadello

Lead Editor: Kim Hamilton Duffy

Related Papers: Signature Format Alignment

About Rebooting the Web of Trust

This paper was produced as part of the Rebooting the Web of Trust IV design workshop. On April 19th through April 21st, 2017, over 40 tech visionaries came together in Paris, France to talk about the future of decentralized trust on the internet with the goal of writing 3-5 white papers and specs. This is one of them.

Workshop Sponsors: Ævatar, Blockstream, Digital Contract Design, Microsoft, Protocol Labs, U Change

Workshop Producer: Christopher Allen

Workshop Facilitators: Christopher Allen and Betty Dhamers, graphic recording by Benoit Pacaud, additional paper editorial & layout by Shannon Appelcline.

What's Next?

The design workshop and this paper are just starting points for Rebooting the Web of Trust. If you have any comments, thoughts, or expansions on this paper, please post them to our GitHub issues page:

https://github.com/WebOfTrustInfo/rebooting-the-web-of-trust-spring2017/issues

The next Rebooting the Web of Trust design workshop is scheduled for Fall 2017 in Boston, Massachusetts. If you'd like to be involved or would like to help sponsor these events, email:

ChristopherA@LifeWithAlacrity.com