Workgroup: jose
Internet-Draft:
draft-jmiller-jose-json-proof-algorithms-01
Published: 10 March 2023
Intended Status: Standards Track
Expires: 11 September 2023
Authors: J. Miller
Ping Identity
M. Jones
Microsoft

JSON Proof Algorithms

Abstract

The JSON Proof Algorithms (JPA) specification registers cryptographic algorithms and identifiers to be used with the JSON Web Proof (JWP) and JSON Web Key (JWK) specifications. It defines several IANA registries for these identifiers.

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 11 September 2023.

Copyright Notice

Table of Contents

1. Introduction

The JSON Web Proof (JWP) draft establishes a new secure container format that supports selective disclosure and unlinkability using Zero-Knowledge Proofs (ZKPs) or other cryptographic algorithms.

> Editor's Note: This draft is still early and incomplete, there will be significant changes to the algorithms as currently defined here. Please do not use any of these definitions or examples for anything except personal experimentation and learning. Contributions and feedback are welcome at https://github.com/json-web-proofs/json-web-proofs.

2. Conventions and Definitions

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.

The roles of "issuer", "holder", and "verifier" are used as defined by the Verifiable Credentials Data Model v1.1. The term "presentation" is also used as defined by this source, but the term "credential" is avoided in this specification in order to minimize confusion with other definitions.

3. Terminology

The terms "JSON Web Signature (JWS)", "Base64url Encoding", "Header Parameter", "JOSE Header", "JWS Payload", "JWS Signature", and "JWS Protected Header" are defined by [RFC7515].

The terms "JSON Web Proof (JWP)", "JWP Payload", "JWP Proof", and "JWP Protected Header" are defined by the JWP draft.

These terms are defined by this specification:

Stable Key An asymmetric key-pair used by an issuer that is also shared via an out-of-band mechanism to a verifier in order to validate the signature.

Ephemeral Key An asymmetric key-pair that is generated for one-time use by an issuer and never stored or used again outside of the creation of a single JWP.

Presentation Key An asymmetric key-pair that is generated by a holder and used to ensure that a presentation is not able to be replayed by any other party.

4. Background

JWP defines a container binding together a protected header, one or more payloads, and a cryptographic proof. It does not define any details about the interactions between an application and the cryptographic libraries that implement proof-supporting algorithms.

Due to the nature of ZKPs, this specification also documents the subtle but important differences in proof algorithms versus those defined by the JSON Web Algorithms [RFC7518]. These differences help support more advanced capabilities such as blinded signatures and predicate proofs.

5. Algorithm Basics

The four principal interactions that every proof algorithm MUST support are [issue](#issue), [confirm](#confirm), [present](#present), and [verify](#verify).

5.1. Issue

The JWP is first created as the output of a JPA's issue operation.

Every algorithm MUST support a JSON issuer protected header along with one or more octet string payloads. The algorithm MAY support using additional items provided by the holder for issuance such as blinded payloads, keys for replay prevention, etc.

All algorithms MUST provide integrity protection for the issuer header and all payloads and MUST specify all digest and/or hash2curve methods used.

5.2. Confirm

Performed by the holder to validate the issued JWP is correctly formed and protected.

Each algorithm MAY support using additional input items options such as those sent to the issuer for issuance. After confirmation, an algorithm MAY return a modified JWP for serialized storage without the local state (such as with blinded payloads now unblinded).

The algorithm MUST fully verify the issued proof value against the issuer protected header and all payloads. If given a presented JWP instead of an issued one the confirm process MUST return an error.

5.3. Present

Used to apply any selective disclosure choices and perform any unlinkability transformations.

An algorithm MAY support additional input options from the requesting party such as for predicate proofs and verifiable computation requests.

Every algorithm MUST support the ability to hide any or all payloads. It MUST always include the issuer protected header unmodified in the presentation.

The algorithm MUST replace the issued proof value and generate a new presented proof value. It also MUST include a new presentation protected header that provides replay protection.

5.4. Verify

Performed by the verifier to verify the protected headers along with any disclosed payloads and/or assertions about them from the proving party, while also verifying they are the same payloads and ordering as witnessed by the issuer.

The algorithm MUST verify the integrity of all disclosed payloads and MUST also verify the integrity of both the issuer and presentation protected headers.

If the presented proof contains any assertions about the hidden payloads, the algorithm MUST also verify all of those assertions. It MAY support additional options such as those sent to the holder to generate the presentation.

If given an issued JWP for verification, the algorithm MUST return an error.

6. Algorithm Specifications

This section defines how to use specific algorithms for JWPs.

6.1. Single Use

> Editor's Note: This algorithm is going to be renamed and slightly refactored; the new name is still TBD.

The Single Use (SU) algorithm is based on composing multiple traditional JWS values into a single JWP proof value. It enables a very simple form of selective disclosure without requiring any advanced cryptographic techniques.

It does not support unlinkability if the same JWP is presented multiple times, therefore when privacy is required the holder will need to interact with the issuer again to receive new single-use JWPs (dynamically or in batches).

6.1.1. JWS Algorithm

The Single Use algorithm is based on using multiple JWS values, all of which are generated with the same JSON Web Algorithm (JWA) for signing. This JWA identifier is included as part of the Single Use identifier for JWP.

The chosen JWA MUST be an asymmetric signing algorithm so that each signature can be verified without sharing any private values between the parties. This ensures that the verifier cannot brute force any non-disclosed payloads based only on their disclosed individual signatures.

6.1.2. Holder Setup

In order to support the protection of a presentation by a holder to a verifier, the holder MUST use a Presentation Key during the issuance and the presentation of every Single Use JWP. This Presentation Key MUST be generated and used for only one JWP.

The issuer MUST verify that the holder has possession of this key. The holder-issuer communication to exchange this information is out of scope of this specification but can be easily accomplished by the holder using this key to generate a JWS that signs a value the issuer can verify as unique.

6.1.3. Issuer Setup

To create a Single Use JWP the issuer first generates a unique Ephemeral Key using the selected JWS algorithm. This key-pair will be used to sign each of the payloads of a single JWP and then discarded.

6.1.4. Using JWS

JSON Web Signatures are used to create all of the signature values used by the SU algorithm. This allows an implementation to use an existing JWS library directly for all necessary cryptographic operations without requiring any additional primitives.

Each individual JWS uses a fixed protected header containing only the minimum required alg value. Since this JWS protected header itself is the same for every JWS, it SHOULD be a static value in the form of {"alg":"***"} where *** is the JWA asymmetric signing key algorithm identifier being used. This value is recreated by a verifier using

the correct JWA algorithm value included in the SU algorithm identifier.

If an implementation uses an alternative JWS protected header than this fixed value, a base64url encoded serialized form of the alternate fixed header MUST be included using the jws_header claim in the issuer protected header.

6.1.5. Issuer Protected Header

The JWK of the issuer's Ephemeral Key MUST be included in the issuer protected header with the property name of proof_jwk and contain only the REQUIRED values to represent the public key.

The holder's Presentation Key JWK MUST be included in issuer protected header using the presentation_jwk claim.

The final issuer protected header is then used directly as the body of a JWS and signed using the issuer's Stable Key. The resulting JWS signature value unencoded octet string is the first value in the JWP proof.

6.1.6. Payloads

Each JWP payload is processed in order and signed as a JWS body using the issuer's Ephemeral Key. The resulting JWS signature value unencoded octet string is appended to the JWP proof.

The proof value as an octet string will have a total length that is the sum of the fixed length of the issuer protected header signature plus the fixed length of each of the payload Ephemeral Key signatures. For example, the signature for the ES256 algorithm is 64 octets and for a JWP with five payloads the total proof value length would be 64 * (1 + 5) = 384 octets).

6.1.7. Presentation Protected Header

In order to generate a new presentation, the holder first creates a presentation protected header that is specific to the verifier being presented to. This header MUST contain a claim that both the holder and verifier trust as being unique and non-replayable.

This specification registers a nonce claim for the presentation protected header that contains a string value either generated by the verifier or derived from values provided by the verifier. When present, the verifier MUST ensure the nonce value matches during verification.

The presentation protected header MAY contain other claims that are either provided by the verifier or by the holder. These presentation

claims SHOULD NOT contain values that are common across multiple presentations and SHOULD be unique to a single presentation and verifier.

6.1.8. Presentation

> Editor's Note: The current definition here is incomplete, the holder's signature needs to also incorporate the presented proof.

The holder derives a new proof value when presenting it to a verifier. The presented proof value will always contain the issuer's Stable Key signature for the issuer protected header as the first element.

The second element of the presented proof value is always the holder's Presentation Key signature of the presentation protected header, constructed identically to the issuer protected header by using the serialized JSON value octet string as the JWS body. Signing only the presentation header with the Presentation Key is sufficient to protect the entire presentation since that key is private to the holder and only the contents of the presentation header are used for replay prevention.

The two header signatures are then followed by only the issuer's Ephemeral Key signatures for each payload that is disclosed. The order of the payload signatures is preserved and MUST be in the same order as the included disclosed payloads in the presented JWP. Non-disclosed payloads will NOT have a signature value included. For example, if the second and fifth payloads are hidden then the holder's derived proof value would be of the length 64 * (1 + 1 + the 1st, 2nd, and 4th payload signatures) = 320 octets.

Since the individual signatures in the proof value are unique and remain unchanged across multiple presentations, a Single Use JWP SHOULD only be presented a single time to each verifier in order for the holder to remain unlinkable across multiple presentations.

6.1.9. Verification

The verifier MUST verify the issuer protected header against the first matching JWS signature part in the proof value using the issuer's Stable Key. It MUST also verify the presentation protected header against the second JWS signature part in the proof value using the holder's Presentation Key as provided in the presentation_jwk claim in the issuer protected header.

With the headers verified, the issuer's Ephemeral Key as given in the issuer protected header proof_jwk claim can then be used to verify each of the disclosed payload signatures.

6.1.10. JPA Registration

Proposed JWP alg value is of the format "SU-" appended with the relevant JWS alg value for the chosen public and ephemeral key-pair algorithm, for example "SU-ES256".

6.1.11. Example

See the example in the appendix of the JSON Web Proof draft.

6.2. BBS

The BBS Signature Scheme under active standards development as a work item within the DIF Applied Cryptography Working Group. Prior to this effort, a V1 implementation of BBS has been released and maintained by a community of individuals with notable adoption in multiple early stage decentralized identity projects.

This JSON Proof Algorithm definition for BBS is based on the already released implementation and relies on the provided software API. A future definition with a different alg value will be created to succeed this version as the BBS standardization effort progresses.

This algorithm supports both selective disclosure and unlinkability, enabling the holder to generate multiple presentations from one issued JWP without any verifier being able to correlate those presentations together.

6.2.1. BLS Curve

The pairing friendly elliptic curve used for the BBS software implementation is part of the BLS family with an embedding degree of 12 over a 381-bit prime field. For this JPA, only the group G2 is used.

In the implementation the method used to generate the key pairs is generateBls12381G2KeyPair().

6.2.2. Messages

BBS is a multi-message scheme and operates on an array of individual messages for signing and proof generation. Each message is a single binary octet string. The BBS implementation uses a hash-to-curve method to map each message to a point.

6.2.3. Issuer Protected Header

The UTF-8 octet string of the issuer protected header is the first message in the input array at index 0.

6.2.4. Payloads

The octet strings of each payload are placed into the BBS message array following the issuer protected header message. For example, the first payload is at index 1 of the array and the last payload is always the last message in the array.

In future versions of this algorithm, there will be additional methods defined for transforming a payload into a point such that additional Zero-Knowledge Proof types can be supported by the holder such as range and membership predicates.

6.2.5. Issuance

The issuer's BLS12-381 G2 Stable Key is used to sign the completed message array input containing the octet strings of the issuer protected header and every payload. The result is a signature octet string that is used as the initial JWP proof value.

In the implementation, the method used to perform the signing is blsSign({keyPair, [header, payload1, payload2, ...]}) and returns a binary signature value.

6.2.6. Presentation

The holder must decode the issuer protected header and payload values in order to generate the identical message array that the issuer used.

To generate a presented JWP for a verifier, the holder must use a cryptographic nonce that is provided by that verifier as input. This nonce MUST be a 32-byte octet string that the verifier generated by a secure RNG. How this nonce value is communicated to the holder is out of scope of this presentation. The nonce claim in the presentation protected header is used to store the verifier's given nonce value.

The holder also applies selective disclosure preferences by creating an array of indices of which messages in the input array are to be revealed to the verifier. The revealed indices MUST include the value 0 so that the issuer protected header message is always revealed to the verifier.

The result of creating a proof is an octet string that is used as the presented JWP proof value.

In the implementation, the method used to generate the proof is blsCreateProof({signedProof, publicKey, [issuer_header, payload1, payload2, ...], presentation_header, [0, 2, ...]).

6.2.7. Verification

The verifier decodes the JWP issuer protected header and payload values into a messages array, skipping any non-revealed payloads. The current BBS implementation embeds the revealed indices into the output proof value, so the verification messages array only needs to include the disclosed messages.

In the implementation, the method used to verify the proof is blsVerifyProof({verifyProof, publicKey, [issuer_header, payload2, ...], presentation_header).

6.2.8. JPA Registration

Proposed JWP alg value for BBS based on the software implementation is "BBS-X".

6.2.9. Example

The following example uses the given BLS12-384 key-pair:

Public:

```
[179, 209, 122, 60, 230, 37, 188, 86, 19, 19, 4, 36, 240, 230, 79,
178, 230, 147, 9, 60, 239, 41, 233, 167, 190, 252, 154, 35, 39, 201,
238, 73, 77, 228, 20, 47, 109, 174, 15, 168, 187, 145, 126, 85, 83,
151, 48, 30, 13, 237, 92, 179, 124, 181, 211, 204, 187, 222, 229,
234, 182, 94, 60, 157, 19, 148, 162, 48, 185, 134, 177, 168, 68,
115, 167, 48, 92, 181, 168, 53, 52, 246, 201, 112, 103, 23, 159,
138, 225, 13, 165, 171, 251, 112, 163, 96]
```

Figure 1: bbs-issuer-public-octets

Private:

```
[72, 125, 227, 97, 150, 148, 186, 145, 110, 46, 135, 232, 104, 204,
128, 242, 73, 151, 72, 162, 0, 54, 139, 146, 221, 137, 34, 74, 1,
42, 140, 206]
```

Figure 2: bbs-issuer-private-octets

The protected header used is:

```
{
  "iss": "https://issuer.example",
  "claims": [
    "family_name",
    "given_name",
    "email",
    "age"
  ],
  "typ": "JPT",
  "alg": "BBS-X"
}
```

Figure 3: bbs-issuer-protected-header

The first payload is the string "Doe" with the octet sequence of [34, 68, 111, 101, 34] and base64url-encoded as IkRvZSI.

The second payload is the string "Jay" with the octet sequence of [34, 74, 97, 121, 34] and base64url-encoded as IkpheSI.

The third payload is the string "jaydoe@example.org" with the octet sequence of [34, 106, 97, 121, 100, 111, 101, 64, 101, 120, 97, 109, 112, 108, 101, 46, 111, 114, 103, 34] and base64url-encoded as ImpheWRvZUBleGFtcGxlLm9yZyI.

The fourth payload is the string 42 with the octet sequence of [52, 50] and base64url-encoded as NDI.

The message array used as an input to the BLS implementation is:

```
[
 [123, 34, 105, 115, 115, 34, 58, 34, 104, 116, 116, 112, 115, 58,
  47, 47, 105, 115, 115, 117, 101, 114, 46, 101, 120, 97, 109, 112,
  108, 101, 34, 44, 34, 99, 108, 97, 105, 109, 115, 34, 58, 91, 34,
  102, 97, 109, 105, 108, 121, 95, 110, 97, 109, 101, 34, 44, 34,
  103, 105, 118, 101, 110, 95, 110, 97, 109, 101, 34, 44, 34, 101,
  109, 97, 105, 108, 34, 44, 34, 97, 103, 101, 34, 93, 44, 34, 116,
  121, 112, 34, 58, 34, 74, 80, 84, 34, 44, 34, 97, 108, 103, 34, 58,
  34, 66, 66, 83, 45, 88, 34, 125],
 [ 34, 68, 111, 101, 34 ],
 [ 34, 74, 97, 121, 34 ],
 [34, 106, 97, 121, 100, 111, 101, 64, 101, 120, 97, 109, 112, 108,
 101, 46, 111, 114, 103, 34], [ 52, 50 ]
]
```

Figure 4: bbs-issuer-messages

Using the above inputs, the output of the blsSign() call is the octet string:

```
[180, 3, 66, 254, 9, 205, 20, 88, 175, 82, 90, 34, 26, 178, 80, 225,
91, 209, 120, 23, 185, 159, 76, 73, 189, 236, 115, 141, 31, 83, 43,
42, 186, 247, 196, 236, 70, 19, 123, 80, 249, 146, 237, 172, 48,
208, 193, 62, 100, 59, 154, 22, 52, 165, 184, 250, 71, 52, 106, 233,
26, 240, 251, 214, 122, 133, 61, 241, 70, 127, 83, 240, 112, 130,
181, 151, 160, 214, 43, 213, 83, 211, 238, 191, 1, 65, 135, 147,
226, 197, 24, 104, 183, 9, 141, 207, 21, 106, 136, 161, 115, 142, 3,
196, 155, 52, 174, 205, 212, 13, 174, 220]
```

Figure 5: bbs-issuer-signature

The resulting signed JWP in JSON serialization is:


```
{
  "protected": "eyJpc3MiOiJodHRwczovL2lzc3Vlci5leGFtcGxlIiwiY2xhaW1z
  IjpbImZhbWlseV9uYW1lIiwiZ2l2ZW5fbmFtZSIsImVtYWlsIiwiYWdlIl0sInR5cC
  I6IkpQVCIsImFsZyI6IkJCUy1YIn0",
  "payloads": [
    "IkRvZSI",
    "IkpheSI",
    "ImpheWRvZUBleGFtcGxlLm9yZyI",
    "NDI"
  ],
  "proof": "tANC_gnNFFivUloiGrJQ4VvReBe5n0xJvexzjR9TKyq698TsRhN7UPmS
  7aww0ME-ZDuaFjSluPpHNGrpGvD71nqFPfFGf1PwcIK1l6DWK9VT0-6_AUGHk-LFGG
  i3CY3PFWqIoXOOA8SbNK7N1A2u3A"
}
```


Figure 6: bbs-issued-jwp

The same JWP in compact serialization:


```
ImV5SnBjM01pT2lKb2RIUndjem92TDJsemMzVmxjaTVsZUdGdGNHeGxJaXdpWTJ4aGFX
MXpJanBiSW1aaGJXbHNlVjl1WVcxbElpd2laMmwyWlc1ZmJtRnRaU0lzSW1WdFlXbHNJ
aXdpWVdkbElsMHNJblI1Y0NJNklrcFFWQ0lzSW1Gc1p5STZJa0pDVXkxWUluMCI.IkRv
ZSI~IkpheSI~ImpheWRvZUBleGFtcGxlLm9yZyI~NDI.tANC_gnNFFivUloiGrJQ4VvR
eBe5n0xJvexzjR9TKyq698TsRhN7UPmS7aww0ME-ZDuaFjSluPpHNGrpGvD71nqFPfFG
f1PwcIK1l6DWK9VT0-6_AUGHk-LFGGi3CY3PFWqIoXOOA8SbNK7N1A2u3A
```


Figure 7: bbs-issued-compact

For verification, a nonce is needed:


```
[137, 103, 248, 147, 211, 133, 97, 190, 130, 157, 110, 64, 244, 250,
100, 151, 7, 36, 164, 109, 146, 195, 190, 75, 32, 255, 6, 128, 44,
128, 96, 9]
```

Figure 8: bbs-present-nonce

To generate a proof, the blsCreateProof() method is used with a revealed indexes array argument of [0, 2, 4] and results in the octet string:


```
[0, 5, 21, 169, 73, 242, 49, 111, 234, 26, 186, 194, 204, 174, 241,
30, 165, 50, 117, 236, 144, 95, 147, 186, 219, 190, 135, 205, 66,
179, 227, 86, 151, 246, 174, 234, 204, 46, 171, 249, 225, 198, 135,
81, 131, 225, 141, 217, 47, 217, 127, 176, 15, 98, 110, 233, 74,
220, 230, 27, 201, 117, 114, 211, 41, 183, 44, 64, 185, 45, 140,
153, 49, 73, 199, 93, 208, 248, 212, 175, 106, 199, 83, 255, 128,
77, 152, 250, 166, 101, 78, 248, 10, 106, 236, 24, 238, 21, 34, 134,
128, 186, 132, 153, 123, 86, 88, 156, 246, 203, 23, 253, 248, 217,
233, 1, 168, 208, 12, 193, 222, 142, 90, 28, 223, 241, 130, 164,
144, 83, 0, 15, 165, 25, 156, 145, 243, 39, 88, 249, 246, 185, 152,
3, 220, 72, 180, 0, 0, 0, 116, 133, 180, 58, 53, 105, 120, 124, 227,
160, 78, 229, 74, 209, 111, 164, 101, 183, 86, 122, 212, 126, 90,
23, 228, 109, 184, 73, 75, 114, 177, 142, 178, 89, 107, 100, 189,
187, 74, 143, 167, 218, 186, 193, 189, 247, 14, 134, 40, 0, 0, 0, 2,
5, 130, 120, 86, 255, 28, 33, 145, 20, 149, 195, 8, 4, 200, 212,
178, 67, 147, 230, 174, 192, 9, 158, 94, 179, 144, 63, 60, 82, 255,
216, 4, 85, 108, 209, 194, 209, 177, 106, 69, 215, 235, 177, 83,
244, 1, 195, 102, 135, 99, 20, 121, 7, 252, 26, 187, 206, 16, 250,
134, 1, 255, 197, 92, 130, 105, 241, 175, 35, 22, 210, 101, 158,
113, 214, 222, 3, 4, 168, 188, 251, 34, 213, 211, 224, 150, 147, 38,
164, 229, 151, 226, 223, 188, 181, 180, 204, 228, 58, 107, 55, 232,
148, 180, 199, 42, 181, 127, 59, 233, 234, 188, 0, 0, 0, 4, 93, 196,
31, 38, 151, 105, 231, 46, 228, 46, 86, 196, 136, 212, 175, 170, 83,
21, 78, 19, 224, 211, 122, 7, 92, 71, 17, 171, 66, 122, 56, 130, 45,
19, 172, 217, 65, 63, 246, 39, 6, 30, 77, 132, 86, 36, 41, 3, 234,
72, 146, 200, 101, 150, 159, 108, 140, 15, 195, 57, 249, 154, 191,
204, 91, 30, 159, 32, 157, 24, 3, 110, 90, 102, 99, 206, 42, 58, 1,
181, 215, 85, 29, 32, 131, 46, 76, 25, 5, 43, 203, 32, 215, 167,
169, 108, 56, 174, 146, 51, 174, 40, 190, 22, 37, 93, 156, 245,
208, 26, 55, 180, 135, 115, 70, 96, 106, 243, 213, 131, 196, 63,
165, 42, 157, 22, 94, 46]
```


Figure 9: bbs-present-proof

The resulting verifiable JWP in JSON serialization is:

```
{
  "protected": "eyJpc3MiOiJodHRwczovL2lzc3Vlci5leGFtcGxlIiwiY2xhaW1z
  IjpbImZhbWlseV9uYW1lIiwiZ2l2ZW5fbmFtZSIsImVtYWlsIiwiYWdlIl0sInR5cC
  I6IkpQVCIsImFsZyI6IkJCUy1YIn0",
  "payloads": [
    null,
    "IkpheSI",
    null,
    "NDI"
  ],
  "proof": "AAUVqUnyMW_qGrrCzK7xHqUydeyQX5O6276HzUKz41aX9q7qzC6r-eHG
  h1GD4Y3ZL9l_sA9ibulK3OYbyXVy0ym3LEC5LYyZMUnHXdD41K9qx1P_gE2Y-qZlTv
  gKauwY7hUihoC6hJl7Vlic9ssX_fjZ6QGo0AzB3o5aHN_xgqSQUwAPpRmckfMnWPn2
  uZgD3Ei0AAAAdIW0OjVpeHzjoE7lStFvpGW3VnrUfloX5G24SUtysY6yWWtkvbtKj6
  fausG99w6GKAAAAAIFgnhW_xwhkRSVwwgEyNSyQ5PmrsAJnl6zkD88Uv_YBFVs0cLR
  sWpF1-uxU_QBw2aHYxR5B_wau84Q-oYB_8VcgmnxryMW0mWecdbeAwSovPsi1dPglp
  MmpOWX4t-8tbTM5DprN-iUtMcqtX876eq8AAAABF3EHyaXaecu5C5WxIjUr6pTFU4T
  4NN6B1xHEatCejiCLROs2UE_9icGHk2EViQpA-pIkshllp9sjA_DOfmav8xbHp8gnR
  gDblpmY84qOgG111UdIIMuTBkFK8sg16epbDiukjOuKL4WJV2c9dAaN7SHc0ZgavPV
  g8Q_pSqdFl4u"
}
```


Figure 10: bbs-present-jwp

The same JWP in compact serialization:


```
ImV5SnBjM01pT2lKb2RIUndjem92TDJsemMzVmxjaTVsZUdGdGNHeGxJaXdpWTJ4aGFX
MXpJanBiSW1aaGJXbHNlVjl1WVcxbElpd2laMmwyWlc1ZmJtRnRaU0lzSW1WdFlXbHNJ
aXdpWVdkbElsMHNJblI1Y0NJNklrcFFWQ0lzSW1Gc1p5STZJa0pDVXkxWUluMCI.~Ikp
heSI~~NDI.AAUVqUnyMW_qGrrCzK7xHqUydeyQX5O6276HzUKz41aX9q7qzC6r-eHGh1
GD4Y3ZL9l_sA9ibulK3OYbyXVy0ym3LEC5LYyZMUnHXdD41K9qx1P_gE2Y-qZlTvgKau
wY7hUihoC6hJl7Vlic9ssX_fjZ6QGo0AzB3o5aHN_xgqSQUwAPpRmckfMnWPn2uZgD3E
i0AAAAdIW0OjVpeHzjoE7lStFvpGW3VnrUfloX5G24SUtysY6yWWtkvbtKj6fausG99w
6GKAAAAAIFgnhW_xwhkRSVwwgEyNSyQ5PmrsAJnl6zkD88Uv_YBFVs0cLRsWpF1-uxU_
QBw2aHYxR5B_wau84Q-oYB_8VcgmnxryMW0mWecdbeAwSovPsi1dPglpMmpOWX4t-8tb
TM5DprN-iUtMcqtX876eq8AAAABF3EHyaXaecu5C5WxIjUr6pTFU4T4NN6B1xHEatCej
iCLROs2UE_9icGHk2EViQpA-pIkshllp9sjA_DOfmav8xbHp8gnRgDblpmY84qOgG111
UdIIMuTBkFK8sg16epbDiukjOuKL4WJV2c9dAaN7SHc0ZgavPVg8Q_pSqdFl4u
```


Figure 11: bbs-present-compact

6.3. Message Authentication Code

The Message Authentication Code (MAC) JPA uses a MAC to both generate ephemeral keys and compute authentication codes to protect the issuer header and each payload individually.

Like the JWS-based JPA, it also does not support unlinkability if the same JWP is presented multiple times and requires an individually

issued JWP for each presentation in order to fully protect privacy. When compared to the JWS approach, using a MAC requires less computation but can result in potentially larger presentation proof values.

The design is intentionally minimal and only involves using a single standardized MAC method instead of a mix of MAC/hash methods or a custom hash-based construct. It is able to use any published cryptographic MAC method such as HMAC or KMAC. It uses traditional public-key based signatures to verify the authenticity of the issuer and holder.

6.3.1. Holder Setup

Prior to the issuer creating a new JWP it must have presentation binding information provided by the holder. This enables the holder to perform replay prevention while presenting the JWP.

The presentation key used by the holder must be transferred to the issuer and verified, likely through a challenge and self-signing mechanism. If the holder requires unlinkability it must also generate a new key that is verified and bound to each new JWP.

How these holder presentation keys are transferred and verified is out of scope of this specification, protocols such as OpenID Connect can be used to accomplish this. What is required by this definition is that the holder's presentation key MUST be included in the issuer's protected header using the pjwk claim with a JWK as the value.

6.3.2. Issuer Setup

To use the MAC algorithm the issuer must have a stable public key pair to perform signing. To start the issuance process, a single 32-byte random Shared Secret must first be generated. This value will be shared privately to the holder as part of the issuer's JWP proof value.

The Shared Secret is used by both the issuer and holder as the MAC method's key to generate a new set of unique ephemeral keys. These keys are then used as the input to generate a MAC that protects each payload.

6.3.3. Issuer Protected Header

The holder's presentation key JWK MUST be included in the issuer protected header using the pjwk claim. The issuer MUST validate that the holder has possession of this key through a trusted mechanism such as verifying the signature of a unique nonce value from the holder.

For consistency, the issuer header is also protected by a MAC by using the fixed value "issuer_header" as the input key. The issuer header JSON is serialized using UTF-8 and encoded with base64url into an octet array. The final issuer header MAC is generated from the octet array and the fixed key, and the resulting value becomes the first input into the larger octet array that will be signed by the issuer.

6.3.4. Payloads

A unique key is generated for each payload using the MAC with the Shared Secret as the key and the values "payload*X" where "X" is replaced by the zero-based array index of the payload, for example "payload*0", "payload_1", etc.

Each payload is serialized using UTF-8 and encoded with base64url into an octet array. The generated key for that payload based on its index is used to generate the MAC for the payload's encoded octet array. The resulting value is appended to the larger octet array that will be signed by the issuer.

6.3.5. Issuer Proof

The issuer proof consists of two items appended together, the issuer's signature of the appended array of MACs, and the Shared Secret used to generate the set of payload keys.

To generate the signature, the array containing the final MAC of the issuer protected header followed by all of the payload MACs appended in order is used as the input to a new JWS.

```
jws_payload = [issuer_header_mac, payload_mac_1, ... payload_mac_n]
```

The issuer signs the JWS using its stable public key and a fixed header containing the alg associated with MAC algorithm in use.

```
jws_header = '{"alg":"ES256"}'
```

The resulting signature is decoded and used as the first item in the issuer proof value. The octet array of the Shared Secret is appended, resulting in the final issuer proof value.

```
issuer_proof = [jws_signature, shared_secret]
```

6.3.6. Presentation Protected Header

See the JWS Presentation Protected Header section.

6.3.7. Presentation

> Editor's Note: The current definition here is incomplete, the holder's signature needs to also incorporate the presented proof.

The presentation proof is constructed as a large octet array containing multiple appended items similar to the issuer proof value. The first item is the JWS decoded signature value generated when the holder uses the presentation key to sign the presentation header. The second item is the issuer signature from the issuer's proof value.

These two signatures are then followed by a MAC value for each payload. The MAC values used will depend on if that payload has been disclosed or is hidden. Disclosed payloads will include the MAC key input, and hidden payloads will include only their final MAC value.

```
presentation_proof = [presentation_signature, issuer_signature,
                      disclosed_key_0, hidden_mac_1, hidden_mac_2,
                      ... disclosed_key_n]
```

The size of this value will depend on the underlying cryptographic algorithms. For example, MAC-H256 uses the ES256 JWS with a decoded signature of 64 octets, and for a JWP with five payloads using HMAC-SHA256 the total presentation proof value length would be 64 + 64 + (5 * 32) = 288 octets.

6.3.8. Verifier Setup

In order to verify that the presentation was protected from replay attacks, the verifier must be able to validate the presentation protected header. This involves the following steps:

1. JSON parse the presentation header
2. Validate the contained nonce claim
3. JSON parse the issuer header
4. Validate the contained pjwk claim
5. Create a JWS using the correct fixed header with alg value and the presentation header as the body
6. Remove the presentation_signature from the beginning of the presentation_proof octet array
7. Validate the JWS using the JWK from the pjwk claim and the presentation_signature value

Next, the verifier must validate all of the disclosed payloads using the following steps:

1. JSON parse the issuer header

2. Resolve the kid using a trusted mechanism to obtain the correct issuer JWK

3. Remove the issuer_signature from the beginning of the remaining presentation_proof octet array (after the presentation_signature was removed)

4. Perform the MAC on the presented issuer_header value using the "issuer_header" value as the input key

5. Store the resulting value as the first entry in a new jws_payload octet array

6. Iterate on each presented payload (disclosed or hidden)

    1. Extract the next hash value from the remaining presentation_proof octet array

    2. If the payload was disclosed: perform a MAC using the given hash value as the input key and append the result to the jws_payload octet array

    3. If the payload was hidden: append the given hash value to the jws_payload octet array

7. Create a JWS using a header containing the alg parameter along with the generated jws_payload value as the payload

8. Validate the JWS using the resolved issuer JWK and the extracted issuer_signature value

6.3.9. JPA Registration

Proposed JWP alg value is of the format "MAC-" appended with a unique identifier for the set of MAC and signing algorithms used. Below are the initial registrations:

*MAC-H256 uses HMAC SHA-256 as the MAC and ECDSA using P-256 and SHA-256 for the signatures

*MAC-H384 uses HMAC SHA-384 as the MAC and ECDSA using P-384 and SHA-384 for the signatures

*MAC-H512 uses HMAC SHA-512 as the MAC and ECDSA using P-521 and SHA-512 for the signatures

*MAC-K25519 uses KMAC SHAKE128 as the MAC and EdDSA using Curve25519 for the signatures

*MAC-K448 uses KMAC SHAKE256 as the MAC and EdDSA using Curve448 for the signatures

*MAC-H256K uses HMAC SHA-256 as the MAC and ECDSA using secp256k1 and SHA-256 for the signatures

6.3.10. Example

The following example uses the MAC-H256 algorithm.

This is the Signer's stable private key in the JWK format:


```
{
  "crv": "P-256",
  "kty": "EC",
  "x": "ONebN43-G5DOwZX6jCVpEYEe0bYd5WDybXAG0sL3iDA",
  "y": "b0MHuYfSxu3Pj4DAyDXabAc0mPjpB1worEpr3yyrft4",
  "d": "jnE0-9YvxQtLJEKcyUHU6HQ3Y9nSDnh0NstYJFn7RuI"
}
```


Figure 12: issuer-private-jwk

This is the Signer's generated Shared Secret:


```
[100, 109, 91, 184, 139, 20, 107, 86, 1, 252, 86, 159, 126, 251,
228, 4, 35, 177, 75, 96, 11, 205, 144, 189, 42, 95, 135, 170, 107,
58, 99, 142]
```


Figure 13: mac-shared-secret

This is the Holder's presentation private key in the JWK format:


```
{
  "crv": "P-256",
  "kty": "EC",
  "x": "oB1TPrE_QJIL61fUOOK5DpKgd8j2zbZJtqpILDTJX6I",
  "y": "3JqnrkucLobkdRuOqZXOP9MMlbFyenFOLyGlG-FPACM",
  "d": "AvyDPl1I4xwjrI2iEOi6DxM9ipJe_h_VUN5OvoKvvW8"
}
```


Figure 14: holder-presentation-jwk

The first MAC is generated using the key issuer_header and the base64url-encoded issuer protected header, resulting in this octet array:

```
[140, 88, 59, 30, 127, 113, 27, 237, 78, 200, 182, 114, 94, 123,
198, 128, 102, 232, 178, 88, 252, 248, 57, 2, 231, 19, 145, 8, 160,
197, 66, 166]
```

Figure 15: mac-issuer-header-mac

The issuer generates an array of derived keys with one for each payload by using the shared secret as the key and the index of the payload as the input:

```
[
 [180, 129, 55, 94, 125, 158, 179, 245, 30, 199, 148, 60, 184, 28,
 197, 123, 231, 232, 95, 91, 65, 74, 38, 242, 253, 96, 67, 44, 40,
 220, 250, 4],
 [143, 172, 182, 156, 184, 138, 228, 172, 215, 26, 175, 137, 137,
 25, 159, 141, 213, 12, 214, 29, 231, 200, 13, 94, 116, 22, 41, 115,
 72, 214, 57, 98],
 [144, 73, 77, 66, 230, 187, 217, 186, 246, 41, 138, 25, 39, 203,
 101, 76, 156, 161, 244, 130, 203, 166, 184, 154, 7, 4, 218, 84,
 168, 199, 36, 245],
 [70, 55, 182, 105, 101, 130, 254, 234, 68, 224, 219, 97, 119, 98,
 244, 33, 43, 55, 148, 238, 225, 177, 101, 160, 49, 246, 109, 155,
 242, 236, 21, 138]
]
```

Figure 16: mac-issuer-keys

The first payload is the string "Doe" with the octet sequence of [34, 68, 111, 101, 34] and base64url-encoded as IkRvZSI.

The second payload is the string "Jay" with the octet sequence of [34, 74, 97, 121, 34] and base64url-encoded as IkpheSI.

The third payload is the string "jaydoe@example.org" with the octet sequence of [34, 106, 97, 121, 100, 111, 101, 64, 101, 120, 97, 109, 112, 108, 101, 46, 111, 114, 103, 34] and base64url-encoded as ImpheWRvZUBleGFtcGxlLm9yZyI.

The fourth payload is the string 42 with the octet sequence of [52, 50] and base64url-encoded as NDI.

A MAC is generated for each payload using the generated key for its given index, resulting in an array of MACs:

```
[
 [156, 53, 90, 125, 139, 226, 60, 168, 100, 220, 79, 255, 8, 87, 28,
 220, 237, 112, 161, 91, 39, 68, 137, 203, 92, 243, 16, 116, 64,
 129, 61, 172],
 [239, 17, 12, 35, 111, 129, 51, 87, 43, 86, 234, 38, 89, 149, 169,
 157, 33, 104, 81, 246, 190, 154, 74, 195, 194, 158, 50, 208, 203,
 203, 249, 237],
 [162, 174, 12, 27, 190, 250, 112, 1, 139, 177, 49, 124, 110, 201,
 83, 233, 14, 109, 60, 253, 121, 184, 126, 121, 26, 138, 5, 214, 97,
 96, 216, 80],
 [61, 109, 78, 172, 255, 189, 67, 83, 247, 65, 234, 128, 30, 47,
 145, 70, 129, 26, 41, 41, 78, 4, 151, 230, 232, 127, 135, 230, 14,
 208, 178, 50]
]
```

Figure 17: mac-issuer-macs

Concatenating the issuer protected header MAC with the array of payload MACs produces a single octet array that is signed using the issuer's stable key, resulting in the following signature:

```
[120, 172, 15, 230, 138, 230, 150, 139, 241, 196, 79, 134, 122, 43,
149, 11, 253, 104, 58, 199, 49, 87, 32, 64, 237, 50, 86, 155, 153,
58, 63, 116, 245, 130, 136, 197, 164, 207, 232, 238, 106, 171, 246,
98, 149, 254, 22, 1, 114, 187, 233, 168, 116, 173, 211, 208, 234,
245, 76, 238, 143, 157, 83, 202]
```

Figure 18: mac-issuer-signature

The original shared secret octet string is then concatenated to the end of the issuer signature octet string and the result is base64url-encoded as the issuer's proof value.

The final issued JWP in JSON serialization is:

```
{
  "payloads": [
    "IkRvZSI",
    "IkpheSI",
    "ImpheWRvZUBleGFtcGxlLm9yZyI",
    "NDI"
  ],
  "issuer": "eyJpc3MiOiJodHRwczovL2lzc3Vlci50bGQiLCJjbGFpbXMiOlsiZmF
  taWx5X25hbWUiLCJnaXZlbl9uYW1lIiwiZW1haWwiLCJhZ2UiXSwidHlwIjoiSlBUI
  iwicGp3ayI6eyJjcnYiOiJQLTI1NiIsImt0eSI6IkVDIiwieCI6Im9CMVRQckVfUUp
  JTDYxZlVPT0s1RHBLZ2Q4ajJ6YlpKdHFwSUxEVEpYNkkiLCJ5IjoiM0pxbnJrdWNMb
  2JrZFJ1T3FaWE9QOU1NbGJGeWVuRk9MeUdsRy1GUEFDTSJ9LCJhbGciOiJNQUMtSDI
  1NiJ9",
  "proof": "eKwP5ormlovxxE-GeiuVC_1oOscxVyBA7TJWm5k6P3T1gojFpM_o7mqr
  9mKV_hYBcrvpqHSt09Dq9Uzuj51TymRtW7iLFGtWAfxWn3775AQjsUtgC82QvSpfh6
  prOmOO"
}
```


Figure 19: mac-issued-jwp

The same JWP in compact serialization:


```
eyJpc3MiOiJodHRwczovL2lzc3Vlci50bGQiLCJjbGFpbXMiOlsiZmFtaWx5X25hbWUi
LCJnaXZlbl9uYW1lIiwiZW1haWwiLCJhZ2UiXSwidHlwIjoiSlBUIiwicGp3ayI6eyJj
cnYiOiJQLTI1NiIsImt0eSI6IkVDIiwieCI6Im9CMVRQckVfUUpJTDYxZlVPT0s1RHBL
Z2Q4ajJ6YlpKdHFwSUxEVEpYNkkiLCJ5IjoiM0pxbnJrdWNMb2JrZFJ1T3FaWE9QOU1N
bGJGeWVuRk9MeUdsRy1GUEFDTSJ9LCJhbGciOiJNQUMtSDI1NiJ9.IkRvZSI~IkpheSI
~ImpheWRvZUBleGFtcGxlLm9yZyI~NDI.eKwP5ormlovxxE-GeiuVC_1oOscxVyBA7TJ
Wm5k6P3T1gojFpM_o7mqr9mKV_hYBcrvpqHSt09Dq9Uzuj51TymRtW7iLFGtWAfxWn37
75AQjsUtgC82QvSpfh6prOmOO
```


Figure 20: mac-issued-compact

Next, we show the presentation of the JWP with selective disclosure.

We start with this presentation header using a nonce provided by the Verifier:


```
{
  "nonce": "uTEB371l1pzWJl7afB0wi0HWUNk1Le-bComFLxa8K-s"
}
```


Figure 21: mac-presentation-header

When signed with the holder's presentation key, the resulting signature octets are:

```
[126, 134, 175, 2, 165, 12, 103, 11, 116, 72, 94, 228, 240, 142,
107, 195, 198, 238, 218, 203, 63, 198, 105, 175, 1, 69, 182, 5, 204,
239, 35, 149, 85, 55, 4, 169, 109, 243, 88, 213, 12, 1, 167, 235,
222, 17, 232, 118, 110, 111, 47, 165, 102, 142, 0, 1, 226, 117, 143,
125, 132, 62, 231, 145]
```

Figure 22: mac-presentation-header-signature

Then by applying selective disclosure of only the given name and age claims (family name and email hidden, payload array indexes 0 and 2), the holder builds a mixed array of either the payload key (if disclosed) or MAC (if hidden):

```
[
 [156, 53, 90, 125, 139, 226, 60, 168, 100, 220, 79, 255, 8, 87, 28,
 220, 237, 112, 161, 91, 39, 68, 137, 203, 92, 243, 16, 116, 64,
 129, 61, 172],
 [143, 172, 182, 156, 184, 138, 228, 172, 215, 26, 175, 137, 137,
 25, 159, 141, 213, 12, 214, 29, 231, 200, 13, 94, 116, 22, 41, 115,
 72, 214, 57, 98],
 [162, 174, 12, 27, 190, 250, 112, 1, 139, 177, 49, 124, 110, 201,
 83, 233, 14, 109, 60, 253, 121, 184, 126, 121, 26, 138, 5, 214, 97,
 96, 216, 80],
 [70, 55, 182, 105, 101, 130, 254, 234, 68, 224, 219, 97, 119, 98,
 244, 33, 43, 55, 148, 238, 225, 177, 101, 160, 49, 246, 109, 155,
 242, 236, 21, 138]
]
```

Figure 23: mac-presentation-keyormac

The final presented proof value is generated by concatenating first the presentation header signature octet string, followed by the issuer signature octet string, then followed by the mixed array of keys and MACs:

```
[126, 134, 175, 2, 165, 12, 103, 11, 116, 72, 94, 228, 240, 142,
107, 195, 198, 238, 218, 203, 63, 198, 105, 175, 1, 69, 182, 5, 204,
239, 35, 149, 85, 55, 4, 169, 109, 243, 88, 213, 12, 1, 167, 235,
222, 17, 232, 118, 110, 111, 47, 165, 102, 142, 0, 1, 226, 117, 143,
125, 132, 62, 231, 145, 120, 172, 15, 230, 138, 230, 150, 139, 241,
196, 79, 134, 122, 43, 149, 11, 253, 104, 58, 199, 49, 87, 32, 64,
237, 50, 86, 155, 153, 58, 63, 116, 245, 130, 136, 197, 164, 207,
232, 238, 106, 171, 246, 98, 149, 254, 22, 1, 114, 187, 233, 168,
116, 173, 211, 208, 234, 245, 76, 238, 143, 157, 83, 202, 156, 53,
90, 125, 139, 226, 60, 168, 100, 220, 79, 255, 8, 87, 28, 220, 237,
112, 161, 91, 39, 68, 137, 203, 92, 243, 16, 116, 64, 129, 61, 172,
143, 172, 182, 156, 184, 138, 228, 172, 215, 26, 175, 137, 137, 25,
159, 141, 213, 12, 214, 29, 231, 200, 13, 94, 116, 22, 41, 115, 72,
214, 57, 98, 162, 174, 12, 27, 190, 250, 112, 1, 139, 177, 49, 124,
110, 201, 83, 233, 14, 109, 60, 253, 121, 184, 126, 121, 26, 138, 5,
214, 97, 96, 216, 80, 70, 55, 182, 105, 101, 130, 254, 234, 68, 224,
219, 97, 119, 98, 244, 33, 43, 55, 148, 238, 225, 177, 101, 160, 49,
246, 109, 155, 242, 236, 21, 138]
```

Figure 24: mac-presentation-proof

The resulting presented JWP in JSON serialization is:


```
{
  "payloads": [
    null,
    "IkpheSI",
    null,
    "NDI"
  ],
  "issuer": "eyJpc3MiOiJodHRwczovL2lzc3Vlci50bGQiLCJjbGFpbXMiOlsiZmF
  taWx5X25hbWUiLCJnaXZlbl9uYW1lIiwiZW1haWwiLCJhZ2UiXSwidHlwIjoiSlBUI
  iwicGp3ayI6eyJjcnYiOiJQLTI1NiIsImt0eSI6IkVDIiwieCI6Im9CMVRQckVfUUp
  JTDYxZlVPT0s1RHBLZ2Q4ajJ6YlpKdHFwSUxEVEpYNkkiLCJ5IjoiM0pxbnJrdWNMb
  2JrZFJ1T3FaWE9QOU1NbGJGeWVuRk9MeUdsRy1GUEFDTSJ9LCJhbGciOiJNQUMtSDI
  1NiJ9",
  "proof": "foavAqUMZwt0SF7k8I5rw8bu2ss_xmmvAUW2BczvI5VVNwSpbfNY1QwB
  p-veEeh2bm8vpWaOAAHidY99hD7nkXisD-aK5paL8cRPhnorlQv9aDrHMVcgQO0yVp
  uZOj909YKIxaTP6O5qq_Zilf4WAXK76ah0rdPQ6vVM7o-dU8qcNVp9i-I8qGTcT_8I
  Vxzc7XChWydEictc8xB0QIE9rI-stpy4iuSs1xqviYkZn43VDNYd58gNXnQWKXNI1j
  lioq4MG776cAGLsTF8bslT6Q5tPP15uH55GooF1mFg2FBGN7ZpZYL-6kTg22F3YvQh
  KzeU7uGxZaAx9m2b8uwVig",
  "presentation": "eyJub25jZSI6InVURUIzNzFsMXB6V0psN2FmQjB3aTBIV1VOa
  zFMZS1iQ29tRkx4YThLLXMifQ"
}
```


Figure 25: mac-presentation-jwp

The same JWP in compact serialization:


```
eyJpc3MiOiJodHRwczovL2lzc3Vlci50bGQiLCJjbGFpbXMiOlsiZmFtaWx5X25hbWUi
LCJnaXZlbl9uYW1lIiwiZW1haWwiLCJhZ2UiXSwidHlwIjoiSlBUIiwicGp3ayI6eyJj
cnYiOiJQLTI1NiIsImt0eSI6IkVDIiwieCI6Im9CMVRQckVfUUpJTDYxZlVPT0s1RHBL
Z2Q4ajJ6YlpKdHFwSUxEVEpYNkkiLCJ5IjoiM0pxbnJrdWNMb2JrZFJ1T3FaWE9QOU1N
bGJGeWVuRk9MeUdsRy1GUEFDTSJ9LCJhbGciOiJNQUMtSDI1NiJ9.eyJub25jZSI6InV
URUIzNzFsMXB6V0psN2FmQjB3aTBIV1VOazFMZS1iQ29tRkx4YThLLXMifQ.~IkpheSI
~~NDI.foavAqUMZwt0SF7k8I5rw8bu2ss_xmmvAUW2BczvI5VVNwSpbfNY1QwBp-veEe
h2bm8vpWaOAAHidY99hD7nkXisD-aK5paL8cRPhnorlQv9aDrHMVcgQO0yVpuZOj909Y
KIxaTP6O5qq_Zilf4WAXK76ah0rdPQ6vVM7o-dU8qcNVp9i-I8qGTcT_8IVxzc7XChWy
dEictc8xB0QIE9rI-stpy4iuSs1xqviYkZn43VDNYd58gNXnQWKXNI1jlioq4MG776cA
GLsTF8bslT6Q5tPP15uH55GooF1mFg2FBGN7ZpZYL-6kTg22F3YvQhKzeU7uGxZaAx9m
2b8uwVig
```


Figure 26: mac-presentation-compact

6.4. ZKSnark

> Editor's Note: This is just a placeholder for a future definition that is in the early stages of development as part of the Decentralized Identity Foundation.

7. Security Considerations

> Editor's Note: This will follow once the algorithms defined here have become more stable.

*Data minimization of the proof value

*Unlinkability of the protected header contents

8. IANA Considerations

8.1. JWP Algorithms Registry

This section establishes the IANA JWP Algorithms Registry. It also registers the following algorithms.

TBD

9. Informative 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>.

[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>.

Appendix A. Acknowledgements

TBD

Appendix B. Document History

[[To be removed from the final specification]]

-01

*Applied editorial improvements

-00

*First individual draft targeting JOSE working group

Authors' Addresses

Jeremie Miller
Ping Identity

Email: jmiller@pingidentity.com

Michael B. Jones
Microsoft

Email: mbj@microsoft.com
URI: https://self-issued.info/