Network Working Group
Internet-Draft
Intended status: Experimental
Expires: 30 September 2025

W. McNally
C. Allen
Blockchain Commons

29 March 2025

The Gordian Envelope Structured Data Format
draft-mcnally-envelope-09

Abstract

Gordian Envelope specifies a structured format for hierarchical binary data focused on the ability to transmit it in a privacy-focused way, offering support for privacy as described in RFC 6973 and human rights as described in RFC 8280. Envelopes are designed to facilitate "smart documents" and have a number of unique features including: easy representation of a variety of semantic structures, a built-in Merkle-like digest tree, deterministic representation using CBOR, and the ability for the holder of a document to selectively elide specific parts of a document without invalidating the digest tree structure. This document specifies the base Envelope format, which is designed to be extensible.

Discussion Venues

This note is to be removed before publishing as an RFC.

Source for this draft and an issue tracker can be found at https://github.com/BlockchainCommons/envelope-internet-draft.

Status of This Memo

Table of Contents

1. Introduction

Gordian Envelope was designed with two key goals in mind: to be _Structure-Ready_, allowing for the reliable and interoperable encoding and storage of information; and to be _Privacy-Ready_, ensuring that transmission of that data can occur in a privacy-protecting manner.

* *Structure-Ready.* Gordian Envelope is designed as a "smart document": a set of information about a subject. More than that, it's a meta-document that can contain or refer to other documents. It can support multiple data structures, from single data items, to simple hierarchies, to labeled property graphs, semantic triples, and other forms of structured graphs. Though its fundamental structure is a tree, it can be used to create Directed Acyclic Graphs (DAGs) through references within or between Envelopes.

* *Privacy-Ready.* Gordian Envelope protects privacy by affording progressive trust, allowing for holders (not just issuers) to minimally disclose information by using elision, and then to optionally increase that disclosure over time. Progressive trust in Gordian Envelopes is accomplished through the hashing of all elements, which also creates foundational support for signing and encryption. This directly addresses the data minimization suggested by "Privacy Considerations for Internet Protocols" [RFC6973] and also addresses topics such as Privacy, Accessibility, Censorship Resistance, Reliability, and Integrity, which are listed as guidelines in "Research into Human Rights Protocol Considerations" [RFC8280].

The following architectural decisions support these goals:

* *Structured Merkle Tree.* A variant of the Merkle Tree [MERKLE] structure is created by hashing the elements in the Envelope into a tree of digests. (In this "structured Merkle Tree", all nodes contain both semantic content and digests, rather than semantic content being limited to leaves.)

* *Deterministic Representation.* There is only one way to encode any semantic representation within a Gordian Envelope. This is accomplished through the use of Deterministic CBOR [DCBOR] and the sorting of the Envelope's assertions into a lexicographic order (not to be confused with sorting a CBOR encoding's map keys). Any Envelope that doesn't follow these strict rules will be rejected; as a result, separate actors assembling envelopes from the same information will converge on the same encoded structure.

1.1. Elision Support

* *Holder-initiated Elision.* Elision can be performed by the Holder of a Gordian Envelope, not just the Issuer.

* *Granular Elision.* Elision can be performed on any data within an Envelope including subjects, predicates and objects of assertions, assertions as a whole, and envelopes as a whole. This allows each entity to elide data as is appropriate for the management of their personal or business risk.

* *Progressive Trust.* The elision mechanics in Gordian Envelopes allow for progressive trust, where increasing amounts of data may be revealed over time.

* *Consistent Hashing.* Even when elided, digests for those parts of the Gordian Envelope remain the same. So constructs such as signatures remain verifiable even for elided documents.

* *Reversible Elision.* Elision can be reversed by the Holder of a Gordian Envelope, which means removed information can be selectively replaced without changing the digest tree.

1.2. Extensions

This document is the base specification for Gordian Envelope, which is stable and useful by itself. However it is also designed to support optional extensions, to be specified in separate documents.

A few such extensions may require adding new Envelope cases: these will extend the Envelope format itself, and will therefore need to be supported by Envelope encoders. Examples include symmetric encryption and compression which (like elision) allow for the transformation of Envelope elements without changing the digest tree.

However, most extensions will be specified by defining the semantics of new subjects, predicates, and objects. Such extensions do not require extending the Envelope format but may be supported by tools. Examples include signatures, public-key encryption, digest

decorrelation, intra- and inter-Envelope references using digests, expression evaluation and distributed function calls, diffing and merging envelopes, and inclusion proofs.

Building on this base specification, we expect a robust ecosystem of extensions to emerge, facilitating a wide variety of 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.

This specification makes use of the following terminology:

byte Used in its now-customary sense as a synonym for "octet".

element Synonymous with "sub-Envelope". An Envelope is a tree of elements.

image The source data from which a cryptographic digest is calculated.

3. Envelope Format Specification

This section is normative and specifies the Gordian Envelope binary format in terms of its CBOR components and their sequencing. The formal language used is the Concise Data Definition Language (CDDL) [RFC8610]. To be considered a well-formed Envelope, a sequence of bytes MUST conform to the Gordian dCBOR deterministic CBOR profile [DCBOR] and MUST conform to the specifications in this section.

An Envelope is a tagged enumerated type with five cases. Here is the entire CDDL specification for the base Envelope format. Each case is discussed in detail below:

```
envelope = #6.200(envelope-content)
envelope-content =
    leaf /
    elided /
    node /
    assertion /
    wrapped

leaf = #6.201(any)

elided = sha256-digest
sha256-digest = bytes .size 32

node = [subject, + assertion-element]
subject = envelope-content
assertion-element = assertion / elided-assertion
elided-assertion = elided           ; MUST represent an assertion.

assertion = { predicate => object }
predicate = envelope-content
object = envelope-content

wrapped = envelope
```

Some of these cases create a hierarchical, recursive structure by including children that are themselves Envelopes. Two of these cases (leaf and elided) have no children. The node case adds one or more assertions to the Envelope, each of which is a child. The assertion case is a predicate/object pair, both of which are children. The wrapped case is used to wrap an entire Envelope including its assertions (its child), so that assertions can be made about the wrapped Envelope as a whole.

3.1. Leaf Case Format

A leaf case is used when the Envelope contains only user-defined CBOR content. It is tagged using #6.201, "encapsulated dCBOR", per [DCBOR].

```
leaf = #6.201(any)
```

The leaf case can be discriminated from other Envelope case arms by the fact that it is the only one that is tagged using #6.201.

To preserve deterministic encoding, authors of application-level data formats based on Envelope MUST only encode CBOR that conforms to dCBOR [DCBOR] in the leaf case. Care must be taken to ensure that leaf dCBOR follows best practices for deterministic encoding, such as clearly specifying when tags for nested structures MUST or MUST NOT be used.

3.2. Elided Case Format

An elided case is used as a placeholder for an element that has been elided. It consists solely of the elided Envelope's digest.

```
elided = sha256-digest
sha256-digest = bytes .size 32
```

The elided case can be discriminated from other Envelope case arms by the fact that it is the only one that is a CBOR byte string and always has a length of 32 bytes.

If the method of producing the digest ever changes, the top-level Envelope tag #6.200 MUST be changed to a new value, and the new method MUST be specified in a new document. This is to ensure that the digest tree remains consistent.

3.3. Node Case Format

A node case is encoded as a CBOR array. A node case MUST be used when one or more assertions are present on the Envelope. A node case MUST NOT be present when there is not at least one assertion.

The first element of the array is the Envelope's subject, followed by one or more assertion-elements, each of which MUST either be an assertion or an elided-assertion.

The assertion-elements MUST appear in ascending lexicographic order by their digest (not to be confused with sorting a CBOR map's keys).

The array MUST NOT contain any assertion-elements with identical digests.

For an Envelope to be valid, any elided-assertion Envelopes in the node assertions MUST, if and when unelided, be found to be actual assertion case Envelopes having the same digest.

```
node = [subject, + assertion-element]
subject = envelope-content
assertion-element = assertion / elided-assertion
elided-assertion = elided           ; MUST represent an assertion.
```

The node case can be discriminated from other Envelope case arms by the fact that it is the only one that is a CBOR array.

3.4. Assertion Case Format

An assertion case is used for each of the assertions on the subject of an Envelope. It is encoded as a CBOR map with exactly one map entry:

* The key of the map entry is the Envelope representing the predicate of the assertion.

* The value of the map entry is the Envelope representing the object of the assertion.

```
assertion = { predicate => object }
predicate = envelope-content
object = envelope-content
```

The assertion case can be discriminated from other Envelope case arms by the fact that it is the only one that is a CBOR map.

3.5. Wrapped Case Format

Assertions make semantic statements about an Envelope’s subject. A wrapped case is used where an Envelope, including all its assertions, should be treated as a single element, e.g. for the purpose of adding assertions to an Envelope as a whole, including its assertions.

```
wrapped = envelope
```

The wrapped case can be discriminated from other Envelope case arms by the fact that it is the only one that is top-level CBOR Envelope, always tagged with #6.200.

4. Computing the Digest Tree

This section specifies how the digests for each of the Envelope cases are computed and is normative. The examples in this section may be used as test vectors.

Each of the five enumerated Envelope cases produces an image which is used as input to a cryptographic hash function to produce the digest of its contents.

The overall digest of an Envelope is the digest of its specific case.

In this section:

* digest(image) is the 32-byte hash produced by running the SHA-256 hash function on the input image.

* The .digest attribute is the digest of the named element computed as specified herein.

* The || operator represents the concatenation of byte sequences.

Note that in the examples below, hexadecimal is shown for readability. All the hexadecimal you see is converted to binary before being hashed.

4.1. Leaf Digest Calculation

The leaf case consists of any CBOR object conforming to dCBOR [DCBOR]. The Envelope image is the CBOR serialization of that object:

digest(cbor)

Example

The CBOR serialization of the plaintext string "Hello" (not including the quotes) is:

```
65            # text(5)
   48656C6C6F # "Hello"
```

The following command line calculates the SHA-256 sum of this sequence:

```
$ echo "6548656C6C6F" | xxd -r -p | shasum --binary --algorithm 256 | \
    awk '{ print $1 }'
4d303dac9eed63573f6190e9c4191be619e03a7b3c21e9bb3d27ac1a55971e6b
```

Using the envelope command line tool [ENVELOPE-CLI], we create an Envelope with this string as the subject and display the Envelope's digest. The digest below matches the one above.

```
$ envelope subject type string "Hello" | envelope digest --hex
4d303dac9eed63573f6190e9c4191be619e03a7b3c21e9bb3d27ac1a55971e6b
```

4.2. Elided Digest Calculation

The elided case declares its digest to be the digest of the Envelope for which it is a placeholder.

Example

If we create the Envelope from the leaf example above, elide it, and then request its digest:

```
$ envelope subject type string "Hello" | envelope elide revealing "" | envelope digest --
hex
4d303dac9eed63573f6190e9c4191be619e03a7b3c21e9bb3d27ac1a55971e6b
```

...we see that its digest is the same as its unelided form:

```
$ envelope subject type string "Hello" | envelope digest --hex
4d303dac9eed63573f6190e9c4191be619e03a7b3c21e9bb3d27ac1a55971e6b
```

4.3. Node Digest Calculation

The Envelope image of the node case is the concatenation of the digest of its subject and the digests of its assertions sorted in ascending lexicographic order.

With a node case, there MUST always be at least one assertion.

```
digest(subject.digest || assertion-0.digest ||
    assertion-1.digest || ... || assertion-n.digest)
```

Example

We create four separate Envelopes and display their digests:

```
$ SUBJECT=‘envelope subject type string "Alice"‘
$ envelope digest --hex $SUBJECT
13941b487c1ddebce827b6ec3f46d982938acdc7e3b6a140db36062d9519dd2f

$ ASSERTION_0=‘envelope subject assertion string "knows" string "Bob"‘
$ envelope digest --hex $ASSERTION_0
78d666eb8f4c0977a0425ab6aa21ea16934a6bc97c6f0c3abaefac951c1714a2

$ ASSERTION_1=‘envelope subject assertion string "knows" string "Carol"‘
$ envelope digest --hex $ASSERTION_1
4012caf2d96bf3962514bcfdcf8dd70c351735dec72c856ec5cdcf2ee35d6a91

$ ASSERTION_2=‘envelope subject assertion string "knows" string "Edward"‘
$ envelope digest --hex $ASSERTION_2
65c3ebc3f056151a6091e738563dab4af8da1778da5a02afcd104560b612ca17
```

We combine the Envelopes into a single Envelope with three assertions:

```
$ ENVELOPE=‘envelope assertion add envelope $ASSERTION_0 $SUBJECT | \
    envelope assertion add envelope $ASSERTION_1 | \
    envelope assertion add envelope $ASSERTION_2‘

$ envelope format $ENVELOPE
"Alice" [
    "knows": "Bob"
    "knows": "Carol"
    "knows": "Edward"
]

$ envelope digest --hex $ENVELOPE
6255e3b67ad935caf07b5dce5105d913dcfb82f0392d4d302f6d406e85ab4769
```

Note that in the Envelope notation representation above, the assertions are sorted alphabetically, with "knows": "Edward" coming last. But internally, the three assertions are ordered by digest in ascending lexicographic order, with "Carol" coming first because its digest starting with 4012caf2 is the lowest, as in the tree formatted display below:

```
$ envelope format --type tree $ENVELOPE
6255e3b6 NODE
    13941b48 subj "Alice"
    4012caf2 ASSERTION
        db7dd21c pred "knows"
        afb8122e obj "Carol"
    65c3ebc3 ASSERTION
        db7dd21c pred "knows"
        e9af7883 obj "Edward"
    78d666eb ASSERTION
        db7dd21c pred "knows"
        13b74194 obj "Bob"
```

To replicate this, we make a list of digests, starting with the subject, and then sort each assertion’s digest in ascending lexicographic order:

```
13941b487c1ddebce827b6ec3f46d982938acdc7e3b6a140db36062d9519dd2f
4012caf2d96bf3962514bcfdcf8dd70c351735dec72c856ec5cdcf2ee35d6a91
65c3ebc3f056151a6091e738563dab4af8da1778da5a02afcd104560b612ca17
78d666eb8f4c0977a0425ab6aa21ea16934a6bc97c6f0c3abaefac951c1714a2
```

We then calculate the SHA-256 digest of the concatenation of these four digests. Note that this is the same digest as the composite Envelope’s digest:

```
echo "13941b487c1ddebce827b6ec3f46d982938acdc7e3b6a140db36062d9519dd2f\
4012caf2d96bf3962514bcfdcf8dd70c351735dec72c856ec5cdcf2ee35d6a91\
65c3ebc3f056151a6091e738563dab4af8da1778da5a02afcd104560b612ca17\
78d666eb8f4c0977a0425ab6aa21ea16934a6bc97c6f0c3abaefac951c1714a2" | \
    xxd -r -p | shasum --binary --algorithm 256 | awk '{ print $1 }'
6255e3b67ad935caf07b5dce5105d913dcfb82f0392d4d302f6d406e85ab4769

$ envelope digest --hex $ENVELOPE
6255e3b67ad935caf07b5dce5105d913dcfb82f0392d4d302f6d406e85ab4769
```

4.4. Assertion Digest Calculation

The Envelope image of the assertion case is the concatenation of the digests of the assertion's predicate and object, in that order:

digest(predicate.digest || object.digest)

Example

We create an assertion from two separate Envelopes and display their digests:

```
$ PREDICATE='envelope subject type string "knows"'
$ envelope digest --hex $PREDICATE
db7dd21c5169b4848d2a1bcb0a651c9617cdd90bae29156baaefbb2a8abef5ba

$ OBJECT='envelope subject type string "Bob"'
$ envelope digest --hex $OBJECT
13b741949c37b8e09cc3daa3194c58e4fd6b2f14d4b1d0f035a46d6d5a1d3f11

$ ASSERTION='envelope subject assertion string "knows" string "Bob"'
$ envelope digest --hex $ASSERTION
78d666eb8f4c0977a0425ab6aa21ea16934a6bc97c6f0c3abaefac951c1714a2
```

To replicate this, we make a list of the predicate digest and the object digest, in that order:

```
db7dd21c5169b4848d2a1bcb0a651c9617cdd90bae29156baaefbb2a8abef5ba
13b741949c37b8e09cc3daa3194c58e4fd6b2f14d4b1d0f035a46d6d5a1d3f11
```

We then calculate the SHA-256 digest of the concatenation of these two digests. Note that this is the same digest as the composite Envelope's digest:

```
echo "db7dd21c5169b4848d2a1bcb0a651c9617cdd90bae29156baaefbb2a8abef5ba\
13b741949c37b8e09cc3daa3194c58e4fd6b2f14d4b1d0f035a46d6d5a1d3f11" | \
    xxd -r -p | shasum --binary --algorithm 256 | awk '{ print $1 }'
78d666eb8f4c0977a0425ab6aa21ea16934a6bc97c6f0c3abaefac951c1714a2

$ envelope digest --hex $ASSERTION
78d666eb8f4c0977a0425ab6aa21ea16934a6bc97c6f0c3abaefac951c1714a2
```

4.5. Wrapped Digest Calculation

The Envelope image of the wrapped case is the digest of the wrapped Envelope:

digest(envelope.digest)

Example

As above, we note the digest of a leaf Envelope is the digest of its CBOR:

```
$ envelope subject type string "Hello" | envelope digest --hex
4d303dac9eed63573f6190e9c4191be619e03a7b3c21e9bb3d27ac1a55971e6b

$ echo "6548656C6C6F" | xxd -r -p | shasum --binary --algorithm 256 | \
    awk '{ print $1 }'
4d303dac9eed63573f6190e9c4191be619e03a7b3c21e9bb3d27ac1a55971e6b
```

Now we note that the digest of a wrapped Envelope is the digest of the wrapped Envelope's digest:

```
$ envelope subject type string "Hello" | \
    envelope subject type wrapped | \
    envelope digest --hex
743a86a9f411b1441215fbbd3ece3de5206810e8a3dd8239182e123802677bd7

$ echo "4d303dac9eed63573f6190e9c4191be619e03a7b3c21e9bb\
3d27ac1a55971e6b" \
    | xxd -r -p | shasum --binary --algorithm 256 | awk '{ print $1 }'
743a86a9f411b1441215fbbd3ece3de5206810e8a3dd8239182e123802677bd7
```

5. Envelope Hierarchy

This section is informative, and describes Envelopes from the perspective of their hierarchical structure and the various ways they can be formatted.

Notionally an Envelope can be thought of as a subject and one or more predicate-object pairs called assertions.

Note that the following example is _not_ CDDL or CBOR diagnostic notation, but "Envelope notation," which is a convenient way to describe the structure of an Envelope:

```
subject [
    predicate0: object0
    predicate1: object1
    ...
    predicateN: objectN
]
```

A concrete example of this might be:

```
"Alice" [
    "knows": "Bob"
    "knows": "Carol"
    "knows": "Edward"
]
```

The notional concept of Envelope is helpful, but not technically accurate because Envelope is implemented structurally as an enumerated type consisting of five cases. This allows actual Envelope instances to be more flexible, for example a "bare assertion" consisting of a predicate-object pair with no subject, which is useful in some situations:

```
"knows": "Bob"
```

More common is the opposite case, a subject with no assertions:

```
"Alice"
```

In Envelopes, there are five distinct "positions" of elements, each of which is itself an Envelope and which therefore produces its own digest:

1. Envelope
2. Subject
3. Assertion
4. Predicate
5. Object

The examples above are printed in Envelope notation, which is designed to make the semantic content of Envelopes human-readable, but it doesn't show the actual digests associated with each of the positions. To see the structure more completely, we can display every element of the Envelope in "Tree Format":

```
6255e3b6 NODE
    13941b48 subj "Alice"
    4012caf2 ASSERTION
        db7dd21c pred "knows"
        afb8122e obj "Carol"
    65c3ebc3 ASSERTION
        db7dd21c pred "knows"
        e9af7883 obj "Edward"
    78d666eb ASSERTION
        db7dd21c pred "knows"
        13b74194 obj "Bob"
```

For easy recognition, Envelope trees only show the first four bytes of each digest, but internally all digests are 32 bytes.

From the above Envelope and its tree, we make the following observations:

* The Envelope is a node case, which has the overall Envelope digest.

* The subject "Alice" has its own digest.

* Each of the three assertions have their own digests

* The predicate and object of each assertion each have their own digests.

* The assertions appear in the structure in ascending lexicographic order by digest, which is the actual order in which they are serialized, and which is distinct from Envelope notation, where they appear sorted alphabetically.

The following subsections present each of the five enumerated Envelope cases in four different output formats:

* Envelope Notation

* Envelope Tree

* CBOR Diagnostic Notation

* CBOR hex

These examples may be used as test vectors. In addition, each subsection starts with the envelope command line [ENVELOPE-CLI] needed to generate the Envelope being formatted.

5.1. Leaf Case

Envelope CLI Command Line

```
envelope subject type string "Alice"
```

Envelope Notation

```
"Alice"
```

Tree

```
13941b48 "Alice"
```

CBOR Diagnostic Notation

```
200(   / envelope /
   201("Alice")   / leaf /
)
```

CBOR Hex

```
d8 c8               # tag(200) envelope
   d8 c9            # tag(201) leaf
      65            # text(5)
         416c696365 # "Alice"
```

5.2. Elided Case

Envelope CLI Command Line

```
envelope subject type string "Alice" | envelope elide revealing ""
```

Envelope Notation

```
ELIDED
```

Tree

```
13941b48 ELIDED
```

CBOR Diagnostic Notation

```
  200(   / envelope /
     h'13941b487c1ddebce827b6ec3f46d982938acdc7e3b6a140db36062d9519dd2f'
  )
```

CBOR Hex

```
  d8 c8                                  # tag(200) envelope
     58 20                               # bytes(32)
        13941b487c1ddebce827b6ec3f46d982938acdc7e3b6a140db36062d9519dd2f
```

5.3. Node Case

Envelope CLI Command Line

```
envelope subject type string "Alice" | envelope assertion add pred-obj string "knows" str
ing "Bob"
```

Envelope Notation

```
"Alice" [
    "knows": "Bob"
]
```

Tree

```
8955db5e NODE
    13941b48 subj "Alice"
    78d666eb ASSERTION
        db7dd21c pred "knows"
        13b74194 obj "Bob"
```

CBOR Diagnostic Notation

```
200(   / envelope /
   [
      201("Alice"),   / leaf /
      {
         201("knows"):   / leaf /
         201("Bob")   / leaf /
      }
   ]
)
```

CBOR Hex

```
   d8 c8                    # tag(200) envelope
      82                    # array(2)
         d8 c9              # tag(201) leaf
            65              # text(5)
               416c696365   # "Alice"
         a1                 # map(1)
            d8 c9           # tag(201) leaf
               65           # text(5)
                  6b6e6f7773 # "knows"
            d8 c9           # tag(201) leaf
               63           # text(3)
                  426f62    # "Bob"
```

5.4. Assertion Case

Envelope CLI Command Line

```
envelope subject assertion string "knows" string "Bob"
```

Envelope Notation

```
"knows": "Bob"
```

Tree

```
78d666eb ASSERTION
    db7dd21c pred "knows"
    13b74194 obj "Bob"
```

CBOR Diagnostic Notation

```
200(   / envelope /
   {
      201("knows"):   / leaf /
      201("Bob")   / leaf /
   }
)
```

CBOR Hex

```
d8 c8                  # tag(200) envelope
   a1                  # map(1)
      d8 c9            # tag(201) leaf
         65            # text(5)
            6b6e6f7773 # "knows"
      d8 c9            # tag(201) leaf
         63            # text(3)
            426f62     # "Bob"
```

5.5. Wrapped Case

Envelope CLI Command Line

```
envelope subject type string "Alice" | envelope subject type wrapped
```

Envelope Notation

```
{
    "Alice"
}
```

Tree

```
2bc17c65 WRAPPED
    13941b48 subj "Alice"
```

CBOR Diagnostic Notation

```
200(   / envelope /
   200(   / envelope /
      201("Alice")   / leaf /
   )
)
```

CBOR Hex

```
d8 c8                  # tag(200) envelope
   d8 c8               # tag(200) envelope
      d8 c9            # tag(201) leaf
         65            # text(5)
            416c696365 # "Alice"
```

6. Reference Implementations

This section is informative.

The current reference implementations of Envelope are written in Swift [ENVELOPE-SWIFT] and Rust [ENVELOPE-RUST].

The envelope command line tool [ENVELOPE-CLI] is also written in Rust.

7. Security Considerations

This section is informative unless noted otherwise.

7.1. CBOR Considerations

Generally, this document inherits the security considerations of CBOR [RFC8949]. Though CBOR has limited web usage, it has received strong usage in hardware, resulting in a mature specification. It also inherits the security considerations of Gordian dCBOR [DCBOR].

7.2. Validation Requirements

Unlike HTML, Envelope is intended to be conservative in both what it encodes _and_ what it accepts as valid. This means that receivers of Envelope-based documents should carefully validate them. Any deviation from the validation requirements of this specification MUST result in the rejection of the entire Envelope. Even after validation, Envelope contents should be treated with due skepticism at the application level.

7.3. Choice of SHA-256 Hash Primitive

Envelope uses the SHA-256 digest algorithm [RFC6234], which is regarded as reliable and widely supported by many implementations in both software and hardware.

7.4. Correlated Digests

Elided Envelopes may in some cases inadvertently reveal information by transmitting digests that may be correlated to known information. In many cases this is of no consequence, but when necessary Envelopes can (when constructed) be "salted" by adding assertions that contain random data. This results in perturbing the digest tree, hence decorrelating it (after elision) from digests whose unelided contents are known.

7.5. RFC 6973 Considerations

"Privacy Considerations for Internet Protocols" [RFC6973] lists threats and guidelines related to privacy in internet protocols. Envelope is intended to help internet protocols easily adopt these considerations. It explicitly addresses the privacy-specific threats of correlation, secondary use, and disclosure by supporting the suggested guideline of Data Minimization.

7.6. RFC 8280 Considerations

"Research into Human Rights Protocol Considerations" [RFC8280] lists guidelines for human rights considerations in internet protocols. Envelope similarly adopts many of the guidelines there, improving privacy and censorship resistance through its hashed elision; and accessibility, heterogeneity support, reliability, and integrity through its fundamental data structures.

8. IANA Considerations

8.1. CBOR Tags

RFC Editor: please replace RFCXXXX with the RFC number of this RFC and remove this note.

IANA [IANACBORTAGS] has assigned the following tag:

Tag	Data Item	Semantics	Specification
200	multiple	Gordian Envelope	[RFCXXXX]

Table 1: CBOR Tag for Envelope

This document uses the tag #6.201 for encapsulated dCBOR, which is defined in [DCBOR].

Points of contact:

* Christopher Allen christophera@blockchaincommons.com (mailto:christophera@blockchaincommons.com)

* Wolf McNally wolf@wolfmcnally.com (mailto:wolf@wolfmcnally.com)

8.2. Media Type

The proposed media type [RFC6838] for Envelope is application/envelope+cbor. The authors understand that this will require this document to become an RFC before the media type can be registered.

* Type name: application

* Subtype name: envelope+cbor

* Required parameters: n/a

* Optional parameters: n/a

* Encoding considerations: binary

* Security considerations: See the previous section of this document

* Interoperability considerations: n/a

* Published specification: This document

* Applications that use this media type: None yet, but it is expected that this format will be deployed in protocols and applications.

* Additional information:

  - Magic number(s): n/a

  - File extension(s): .envelope

  - Macintosh file type code(s): n/a

* Person & email address to contact for further information:

  - Christopher Allen christophera@blockchaincommons.com (mailto:christophera@blockchaincommons.com)

  - Wolf McNally wolf@wolfmcnally.com (mailto:wolf@wolfmcnally.com)

* Intended usage: COMMON

* Restrictions on usage: none

* Author:

  - Wolf McNally wolf@wolfmcnally.com (mailto:wolf@wolfmcnally.com)

* Change controller:

  - The IESG iesg@ietf.org (mailto:iesg@ietf.org)

9. References

9.1. Normative References

[DCBOR] "Gordian dCBOR: A Deterministic CBOR Application Profile", n.d., <https://www.ietf.org/archive/id/draft-mcnally-deterministic-cbor-08.html>.

[ENVELOPE-CLI]
"Envelope Command Line Tool (Rust)", n.d.,
<https://crates.io/crates/bc-envelope-cli>.

[ENVELOPE-RUST]
"Blockchain Commons Gordian Envelope for Rust", n.d.,
<https://crates.io/crates/bc-envelope>.

[ENVELOPE-SWIFT]
"Blockchain Commons Gordian Envelope for Swift", n.d.,
<https://github.com/blockchaincommons/BCSwiftEnvelope>.

[IANACBORTAGS]
IANA, "Concise Binary Object Representation (CBOR) Tags",
<https://www.iana.org/assignments/cbor-tags>.

[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/rfc/rfc2119>.

[RFC6234] Eastlake 3rd, D. and T. Hansen, "US Secure Hash Algorithms (SHA and SHA-based HMAC and HKDF)", RFC 6234, DOI 10.17487/RFC6234, May 2011, <https://www.rfc-editor.org/rfc/rfc6234>.

[RFC6838] Freed, N., Klensin, J., and T. Hansen, "Media Type Specifications and Registration Procedures", BCP 13, RFC 6838, DOI 10.17487/RFC6838, January 2013, <https://www.rfc-editor.org/rfc/rfc6838>.

[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/rfc/rfc8174>.

[RFC8610] Birkholz, H., Vigano, C., and C. Bormann, "Concise Data Definition Language (CDDL): A Notational Convention to Express Concise Binary Object Representation (CBOR) and JSON Data Structures", RFC 8610, DOI 10.17487/RFC8610, June 2019, <https://www.rfc-editor.org/rfc/rfc8610>.

[RFC8949] Bormann, C. and P. Hoffman, "Concise Binary Object Representation (CBOR)", STD 94, RFC 8949, DOI 10.17487/RFC8949, December 2020, <https://www.rfc-editor.org/rfc/rfc8949>.

9.2. Informative References

[MERKLE] "Merkle Tree", n.d., <https://en.wikipedia.org/wiki/Merkle_tree>.

[RFC6973] Cooper, A., Tschofenig, H., Aboba, B., Peterson, J., Morris, J., Hansen, M., and R. Smith, "Privacy Considerations for Internet Protocols", RFC 6973, DOI 10.17487/RFC6973, July 2013, <https://www.rfc-editor.org/rfc/rfc6973>.

[RFC8280] ten Oever, N. and C. Cath, "Research into Human Rights Protocol Considerations", RFC 8280, DOI 10.17487/RFC8280, October 2017, <https://www.rfc-editor.org/rfc/rfc8280>.

Acknowledgments

The authors are grateful to Carsten Bormann for his review and helpful feedback.

Authors' Addresses

Wolf McNally
Blockchain Commons
Email: wolf@wolfmcnally.com

Christopher Allen
Blockchain Commons
Email: christophera@lifewithalacrity.com