Workgroup: Network Working Group
Internet-Draft: draft-hallambaker-mesh-dare
Published: 23 October 2022
Intended Status: Informational
Expires: 26 April 2023
Authors: P. M. Hallam-Baker
ThresholdSecrets.com

Mathematical Mesh 3.0 Part III : Data At Rest Encryption (DARE)

Abstract

This document describes the Data At Rest Encryption (DARE) Envelope and Sequence syntax.

The DARE Envelope syntax is used to digitally sign, digest, authenticate, or encrypt arbitrary content data.

The DARE Sequence syntax describes an append-only sequence of entries, each containing a DARE Envelope. DARE Sequences may support cryptographic integrity verification of the entire data container content by means of a Merkle tree.

[Note to Readers]

Discussion of this draft takes place on the MATHMESH mailing list (mathmesh@ietf.org), which is archived at https://mailarchive.ietf.org/arch/search/?email_list=mathmesh.

This document is also available online at http://mathmesh.com/Documents/draft-hallambaker-mesh-dare.html.

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 26 April 2023.

Copyright Notice

Table of Contents

1. Introduction

This document describes the Data At Rest Encryption (DARE) Envelope and Sequence Syntax. The DARE Envelope syntax is used to digitally sign, digest, authenticate, or encrypt arbitrary message content. The DARE Sequence syntax describes an append-only sequence of data frames, each containing a DARE Envelope that supports efficient incremental signature and encryption.

The DARE Envelope Syntax is based on a subset of the JSON Web Signature [RFC7515] and JSON Web Encryption [RFC7516] standards and shares many fields and semantics. The processing model and data structures have been streamlined to remove alternative means of specifying the same content and to enable multiple data sequences to be signed and encrypted under a single master encryption key without compromise to security.

A DARE Envelope consists of a *Header*, *Payload* and an optional *Trailer*. To enable single pass encoding and decoding, the Header contains all the information required to perform cryptographic processing of the Payload and authentication data (digest, MAC, signature values) **MAY** be deferred to the Trailer section.

A DARE Sequence is an append-only log format consisting of a sequence of frames. Cryptographic enhancements (signature, encryption) may be applied to individual frames or to sets of frames. Thus, a single key exchange may be used to provide a master key to encrypt multiple frames and a single signature may be used to authenticate all the frames in the container up to and including the frame in which the signature is presented.

The DARE Envelope syntax may be used either as a standalone cryptographic message syntax or as a means of presenting a single DARE Sequence frame together with the complete cryptographic context required to verify the contents and decrypt them.

1.1. Encryption and Integrity

A key innovation in the DARE Envelope Syntax is the separation of key exchange and data encryption operations so that a Master Key (MK) established in a single exchange to be applied to multiple data sequences. This means that a single public key operation **MAY** be used to encrypt and/or authenticate multiple parts of the same DARE Envelope or multiple frames in a DARE Sequence.

To avoid reuse of the key and to avoid the need to communicate separate IVs, each octet sequence is encrypted under a different encryption key (and IV if required) derived from the Master Key by means of a salt that is unique for each octet sequence that is encrypted. The same approach is used to generate keys for calculating a MAC over the octet sequence if required. This approach allows encryption and integrity protections to be applied to the envelope payload, to header or trailer fields or to application defined Enhanced Data Sequences in the header or trailer.

1.1.1. Key Exchange

Traditional cryptographic containers describe the application of a single key exchange to encryption of a single octet sequence. Examples include PCKS#7/CMS [RFC2315], OpenPGP [RFC4880] and JSON Web Encryption [RFC7516].

To encrypt data using RSA, the encoder first generates a random encryption key and initialization vector (IV). The encryption key is encrypted under the public key of each recipient to create a per-recipient decryption entry. The encryption key, plaintext and IV are used to generate the ciphertext (figure 1).

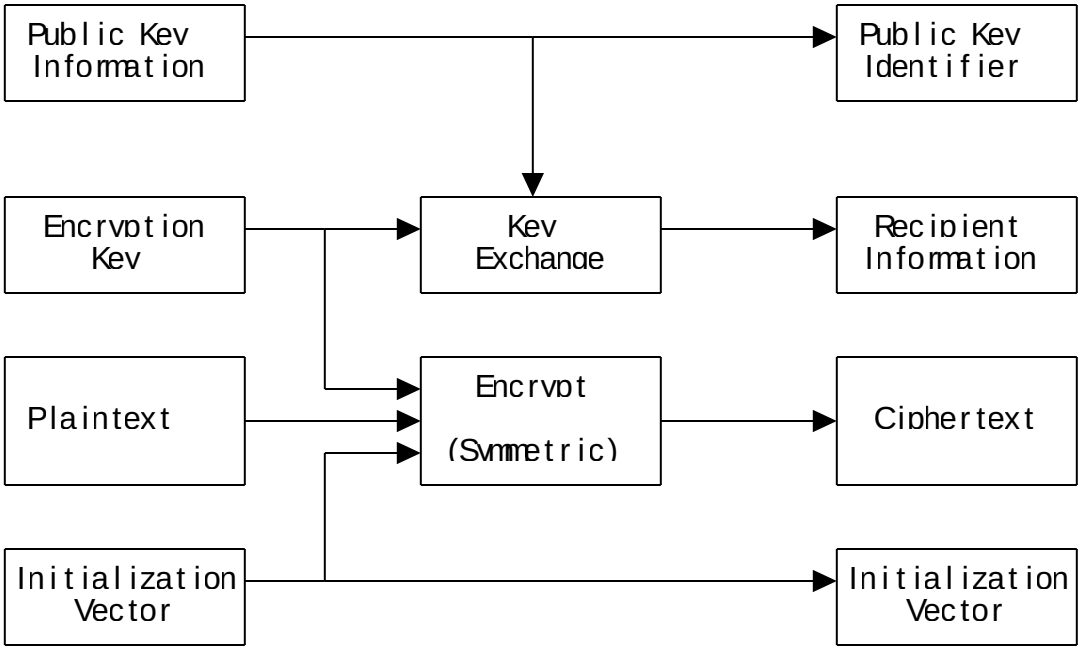

Figure 1: Monolithic Key Exchange and Encrypt

This approach is adequate for the task of encrypting a single octet stream. It is less than satisfactory when encrypting multiple octet streams or very long streams for which a rekeying operation is desirable.

In the DARE approach, key exchange and key derivation are separate operations and keys **MAY** be derived for encryption or integrity purposes or both. A single key exchange **MAY** be used to derive keys to apply encryption and integrity enhancements to multiple data sequences.

The DARE key exchange begins with the same key exchange used to produce the CEK in JWE but instead of using the CEK to encipher data directly, it is used as one of the inputs to a Key Derivation Function (KDF) that is used to derive parameters for each block of data to be encrypted. To avoid the need to introduce additional terminology, the term 'CEK' is still used to describe the output of the key agreement algorithm (including key unwrapping if required) but it is more appropriately described as a Master Key (figure 2).

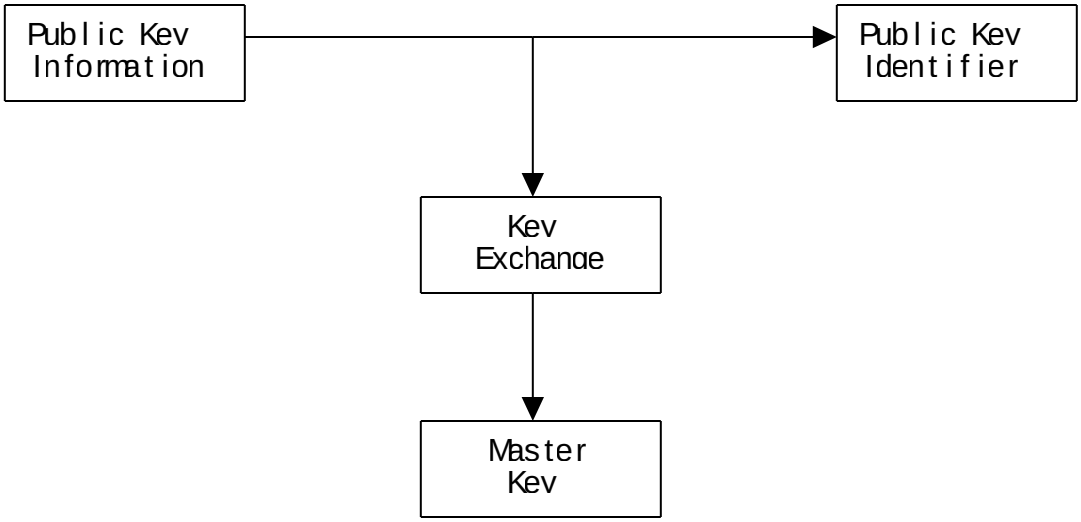

Figure 2: Exchange of Master Key

A Master Key may be used to encrypt any number of data items. Each data item is encrypted under a different encryption key and IV (if required). This data is derived from the Master Key using the HKDF function [RFC5869] using a different salt for each data item and separate info tags for each cryptographic function (figure 3).

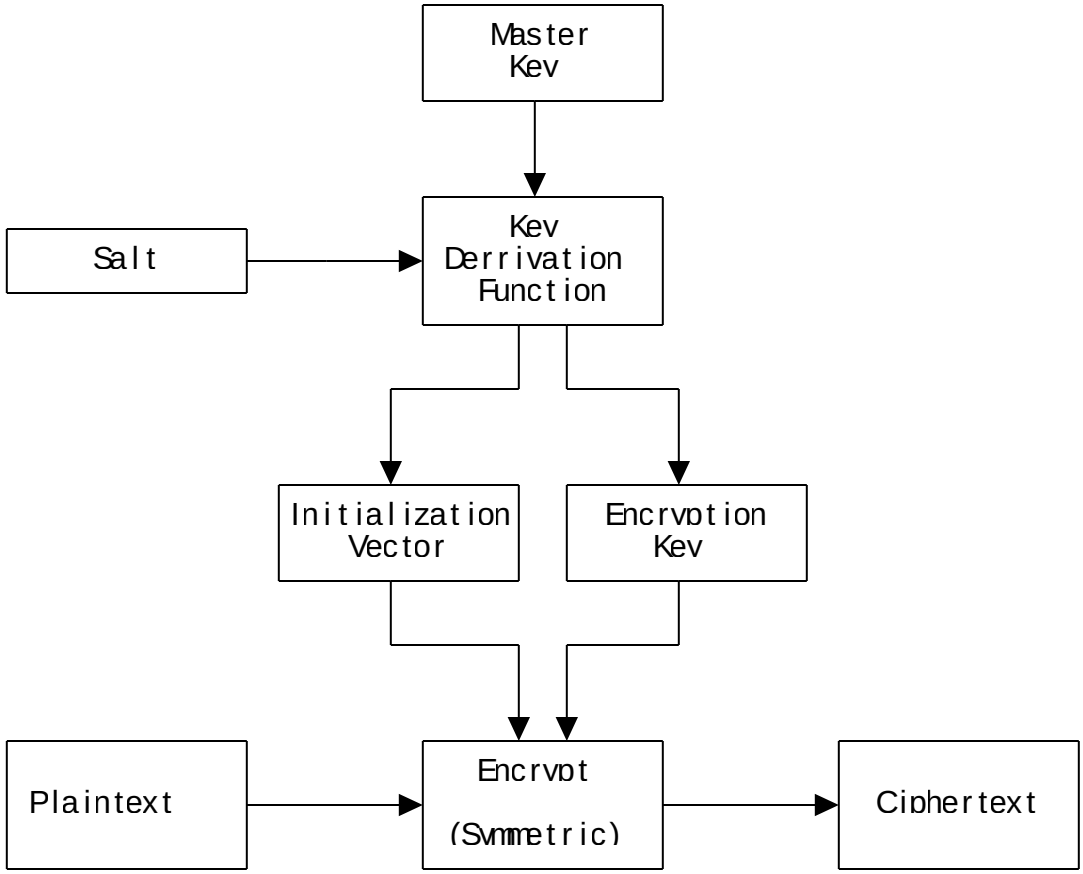

Figure 3: Data item encryption under Master Key and per-item salt.

This approach to encryption offers considerably greater flexibility allowing the same format for data item encryption to be applied at the transport, message or field level.

1.1.2. Data Erasure

Each encrypted DARE Envelope specifies a unique Master Salt value of at least 128 bits which is used to derive the salt values used to derive cryptographic keys for the envelope payload and annotations.

Erasure of the Master Salt value **MAY** be used to effectively render the envelope payload and annotations undecipherable without altering the envelope payload data. The work factor for decryption will be O(2^128) even if the decryption key is compromised.

1.2. Signature

As with encryption, DARE Envelope signatures **MAY** be applied to an individual envelope or a sequence of envelope.

1.2.1. Signing Individual Plaintext Envelopes

When an individual plaintext envelope is signed, the digest value used to create the signature is calculated over the binary value of the payload data. That is, the value of the payload before the encoding (Base-64, JSON-B) is applied.

1.2.2. Signing Individual Encrypted Envelopes

When an individual plaintext envelope is signed, the digest value used to create the signature is calculated over the binary value of the payload data. That is, the value of the payload after encryption but before the encoding (Base-64, JSON-B) is applied.

Use of signing and encryption in combination presents the risk of subtle attacks depending on the order in which signing and encryption take place [Davis2001].

Na?ve approaches in which an envelope is encrypted and then signed present the possibility of a surreptitious forwarding attack. For example, Alice signs an envelope and sends it to Mallet who then strips off Alice's signature and sends the envelope to Bob.

Na?ve approaches in which an envelope is signed and then encrypted present the possibility of an attacker claiming authorship of a ciphertext. For example, Alice encrypts a ciphertext for Bob and then signs it. Mallet then intercepts the envelope and sends it to Bob.

While neither attack is a concern in all applications, both attacks pose potential hazards for the unwary and require close inspection of application protocol design to avoid exploitation.

To prevent these attacks, each signature on an envelope that is signed and encrypted **MUST** include a witness value that is calculated by applying a MAC function to the signature value as described in section XXX.

1.2.3. Signing sequences of envelopes

To sign multiple envelopes with a single signature, we first construct a Merkle tree of the envelope payload digest values and then sign the root of the Merkle tree.

[This is not yet implemented but will be soon.]

1.3. Sequence

DARE Sequence is a message and file syntax that allows a sequence of data frames to be represented with cryptographic integrity,

signature, and encryption enhancements to be constructed in an append only format.

The format is designed to meet the requirements of a wide range of use cases including:

*Recording transactions in persistent storage.

*Synchronizing transaction logs between hosts.

*File archive.

*Message spool.

*Signing and encrypting single data items.

*Incremental encryption and authentication of server logs.

1.3.1. Sequence Format

A Sequence consists of a sequence of variable length Frames. Each frame consists of a forward length indicator, the framed data and a reverse length indicator. The reverse length indicator is written out backwards allowing the length and thus the frame to be read in the reverse direction:

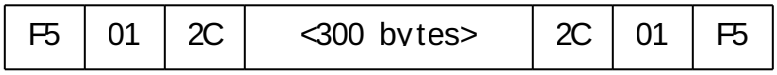

Figure 4: JBCD Bidirectional Frame

Each frame contains a single DARE Envelope consisting of a Header, Payload and Trailer (if required). The first frame in a container describes the container format options and defaults. These include the range of encoding options for frame metadata supported and the container profiles to which the container conforms.

All internal data formats support use of pointers of up to 64 bits allowing containers of up to 18 exabytes to be written.

Five container types are currently specified:

Simple The container does not provide any index or content integrity checks.

Tree Frame headers contain entries that specify the start position of previous frames at the apex of the immediately enclosing

binary tree. This enables efficient random access to any frame in the file.

Digest Each frame trailer contains a PayloadDigest field. Modification of the payload will cause verification of the PayloadDigest value to fail on that frame.

Chain Each frame trailer contains PayloadDigest and ChainDigest fields allowing modifications to the payload data to be detected. Modification of the payload will cause verification of the PayloadDigest value to fail on that frame and verification of the ChainDigest value to fail on all subsequent frames.

Merkle Tree Frame headers contain entries that specify the start position of previous frames at the apex of the immediately enclosing binary tree. Frame Trailers contain TreeDigestPartial and TreeDigestFinal entries forming a Merkle digest tree.

Currently, the Mesh only makes use of the Merkle Tree sequence type.

1.3.2. Write

In normal circumstances, Sequences are written as an append only log. As with Envelopes, integrity information (payload digest, signatures) is written to the entry trailer. Thus, large payloads may be written without the need to buffer the payload data *provided that* the content length is known in advance.

Should exceptional circumstances require, Sequence entries **MAY** be erased by overwriting the Payload and/or parts of the Header content without compromising the ability to verify other entries in the container. If the entry Payload is encrypted, it is sufficient to erase the container salt value to render the container entry effectively inaccessible (though recovery might still be possible if the original salt value can be recovered from the storage media.

1.3.3. Encryption and Authentication

Frame payloads and associated attributes **MAY** be encrypted and/or authenticated in the same manner as Envelopes.

Incremental encryption is supported allowing encryption parameters from a single public key exchange operation to be applied to encrypt multiple frames. The public key exchange information is specified in the first encrypted frame and subsequent frames encrypted under those parameters specify the location at which the key exchange information is to be found by means of the ExchangePosition field which **MUST** specify a location that is earlier in the file.

To avoid cryptographic vulnerabilities resulting from key re-use, the DARE key exchange requires that each encrypted sequence use an encryption key and initialization vector derived from the master key established in the public key exchange by means of a unique salt specified in each envelope.

Each Envelope and by extension, each Sequence frame **MUST** specify a unique salt value of at least 128 bits. Since the encryption key is derived from the salt value by means of a Key Derivation Function, erasure of the salt **MAY** be used as a means of rendering the payload plaintext value inaccessible without changing the payload value.

1.3.4. Integrity and Signature

Signatures **MAY** be applied to a payload digest, the final digest in a chain or tree. The chain and tree digest modes allow a single signature to be used to authenticate all frame payloads in a container.

The tree signature mode is particularly suited to applications such as file archives as it allows files to be verified individually without requiring the signer to sign each individually. Furthermore, in applications such as code signing, it allows a single signature to be used to verify both the integrity of the code and its membership of the distribution.

As with DARE Envelope, the signature mechanism does not specify the interpretation of the signature semantics. The presence of a signature demonstrates that the holder of the private key applied it to the specified digest value but not their motive for doing so. Describing such semantics is beyond the scope of this document and is deferred to future work.

1.3.5. Redaction

The chief disadvantage of using an append-only format is that containers only increase in size. In many applications, much of the data in the container becomes redundant or obsolete and a process analogous to garbage collection is required. This process is called *redaction*.

The simplest method of redaction is to create a new container and sequentially copy each entry from the old container to the new, discarding redundant frames and obsolete header information.

For example, partial index records may be consolidated into a single index record placed in the last frame of the container. Unnecessary signature and integrity data may be discarded and so on.

While redaction could in principle be effected by moving data in-place in the existing container, supporting this approach in a robust fashion is considerably more complex as it requires backward references in subsequent frames to be overridden as each frame is moved.

1.3.6. Alternative approaches

Many file proprietary formats are in use that support some or all of these capabilities but only a handful have public, let alone open, standards. DARE Sequence is designed to provide a superset of the capabilities of existing message and file syntaxes, including:

* Cryptographic Message Syntax [RFC5652] defines a syntax used to digitally sign, digest, authenticate, or encrypt arbitrary message content.

* The.ZIP File Format specification [ZIPFILE] developed by Phil Katz.

* The BitCoin Block chain [BLOCKCHAIN].

* JSON Web Encryption and JSON Web Signature

Attempting to make use of these specifications in a layered fashion would require at least three separate encoders and introduce unnecessary complexity. Furthermore, there is considerable overlap between the specifications providing multiple means of achieving the same ends, all of which must be supported if decoders are to work reliably.

1.3.7. Efficiency

Every data format represents a compromise between different concerns, in particular:

Compactness The space required to record data in the encoding.

Memory Overhead The additional volatile storage (RAM) required to maintain indexes etc. to support efficient retrieval operations.

Number of Operations The number of operations required to retrieve data from or append data to an existing encoded sequence.

Number of Disk Seek Operations Optimizing the response time of magnetic storage media to random access read requests has traditionally been one of the central concerns of database design. The DARE Sequence format is designed to the assumption that this will cease to be a concern as solid state media replaces magnetic.

While the cost of storage of all types has declined rapidly over the past decades, so has the amount of data to be stored. DARE Sequence represents a pragmatic balance of these considerations for current technology. In particular, since payload volumes are likely to be very large, memory and operational efficiency are considered higher priorities than compactness.

2. Definitions

2.1. Related Specifications

The DARE Envelope and Sequence formats are based on the following existing standards and specifications.

Object serialization The JSON-B [draft-hallambaker-jsonbcd] encoding is used for object serialization. This encoding is an extension of the JavaScript Object Notation (JSON) [RFC7159].

Message syntax The cryptographic processing model is based on JSON Web Signature (JWS) [RFC7515], JSON Web Encryption (JWE) [RFC7516] and JSON Web Key (JWK) [RFC7517].

Cryptographic primitives. The HMAC-based Extract-and-Expand Key Derivation Function [RFC5869] and Advanced Encryption Standard (AES) Key Wrap with Padding Algorithm [RFC3394] are used.

The Uniform Data Fingerprint method of presenting data digests is used for key identifiers and other purposes [draft-hallambaker-mesh-udf].

Cryptographic algorithms The cryptographic algorithms and identifiers described in JSON Web Algorithms (JWA) [RFC7518] are used together with additional algorithms as defined in the JSON Object Signing and Encryption IANA registry [IANAJOSE].

2.2. Requirements Language

The key words "**MUST**", "**MUST NOT**", "**REQUIRED**", "**SHALL**", "**SHALL NOT**", "**SHOULD**", "**SHOULD NOT**", "**RECOMMENDED**", "**MAY**", and "**OPTIONAL**" in this document are to be interpreted as described in [RFC2119].

2.3. Defined terms

The terms "Authentication Tag", "Content Encryption Key", "Key Management Mode", "Key Encryption", "Direct Key Agreement", "Key Agreement with Key Wrapping" and "Direct Encryption" are defined in the JWE specification [RFC7516].

The terms "Authentication", "Ciphertext", "Digital Signature", "Encryption", "Initialization Vector (IV)", "Message Authentication

Code (MAC)", "Plaintext" and "Salt" are defined by the Internet Security Glossary, Version 2 [RFC4949].

Annotated Envelope A DARE Envelope that contains an Annotations field with at least one entry.

Authentication Data A Message Authentication Code or authentication tag.

Complete Envelope A DARE envelope that contains the key exchange information necessary for the intended recipient(s) to decrypt it.

Detached Envelope A DARE envelope that does not contain the key exchange information necessary for the intended recipient(s) to decrypt it.

Encryption Context The master key, encryption algorithms and associated parameters used to generate a set of one or more enhanced data sequences.

Encoded data sequence (EDS) A sequence consisting of a salt, content data and authentication data (if required by the encryption context).

Enhancement Applying a cryptographic operation to a data sequence. This includes encryption, authentication and both at the same time.

Generator The party that generates a DARE envelope.

Group Encryption Key A key used to encrypt data to be read by a group of users. This is typically achieved by means of some form of proxy re-encryption or distributed key generation.

Group Encryption Key Identifier A key identifier for a group encryption key.

Master Key (MK) The master secret from which keys are derived for authenticating enhanced data sequences.

Recipient Any party that receives and processes at least some part of a DARE envelope.

Related Envelope A set of DARE envelopes that share the same key exchange information and hence the same Master Key.

Uniform Data Fingerprint (UDF) The means of presenting the result of a cryptographic digest function over a data sequence and

content type identifier specified in the Uniform Data Fingerprint specification [draft-hallambaker-mesh-udf]

3. DARE Envelope Architecture

A DARE Envelope is a sequence of three parts:

Header A JSON object containing information a reader requires to begin processing the envelope.

Payload An array of octets.

Trailer A JSON object containing information calculated from the envelope payload.

For example, the following sequence is a JSON encoded Envelope with an empty header, a payload of zero length and an empty trailer:

```
[ {}, "", {} ]
```

DARE Envelopes **MAY** be encoded using JSON serialization or a binary serialization for greater efficiency.

JSON [RFC7159] Offers compatibility with applications and libraries that support JSON. Payload data is encoded using Base64 incurring a 33% overhead.

JSON-B [draft-hallambaker-jsonbcd] A superset of JSON encoding that permits binary data to be encoded as a sequence of length-data segments. This avoids the Base64 overhead incurred by JSON encoding. Since JSON-B is a superset of JSON encoding, an application can use a single decoder for either format.

JSON-C [draft-hallambaker-jsonbcd] A superset of JSON-C which provides additional efficiency by allowing field tags and other repeated string data to be encoded by reference to a dictionary. Since JSON-C is a superset of JSON and JSON-B encodings, an application can use a single decoder for all three formats.

DARE Envelope processors **MUST** support JSON serialization and **SHOULD** support JSON-B serialization.

3.1. Processing Considerations

The DARE Envelope Syntax supports single pass encoding and decoding without buffering of data. All the information required to begin processing a DARE envelope (key agreement information, digest algorithms), is provided in the envelope header. All the information

that is derived from envelope processing (authentication codes, digest values, signatures) is presented in the envelope trailer.

The choice of envelope encoding does not affect the semantics of envelope processing. A DARE Envelope **MAY** be reserialized under the same serialization or converted from any of the specified serialization to any other serialization without changing the semantics or integrity properties of the envelope.

3.2. Encoded Data Sequence

An encoded data sequence (EDS) is a sequence of octets that encodes a data sequence according to cryptographic enhancements specified in the context in which it is presented. An EDS **MAY** be encrypted and **MAY** be authenticated by means of a MAC. The keys and other cryptographic parameters used to apply these enhancements are derived from the cryptographic context and a Salt prefix specified in the EDS itself.

An EDS sequence contains exactly three binary fields encoded in JSON-B serialization as follows:

Salt Prefix A sequence of octets used to derive the encryption key, Initialization Vector and MAC key as required.

Body The plaintext or encrypted content.

Authentication Tag The authentication code value in the case that the cryptographic context specifies use of authenticated encryption or a MAC, otherwise is a zero-length field.

Requiring all three fields to be present, even in cases where they are unnecessary simplifies processing at the cost of up to six additional data bytes.

The encoding of the 'From' header of the previous example as a plaintext EDS is as follows:

```
88 01
  00
88 17
  46 72 6f 6d 3a 20 41 6c   69 63 65 40 65 78 61 6d
  70 6c 65 2e 63 6f 6d
[EOF]
```

3.3. Content Metadata and Annotations

A header **MAY** contain header fields describing the payload content. These include:

ContentType Specifies the IANA Media Type [RFC6838].

Annotations A list of Encoded Data Sequences that provide application specific annotations to the envelope.

For example, consider the following mail message:

```
From: Alice@example.com
To: bob@example.com
Subject: TOP-SECRET Product Launch Today!

The CEO told me the product launch is today. Tell no-one!
```

Existing encryption approaches require that header fields such as the subject line be encrypted with the body of the message or not encrypted at all. Neither approach is satisfactory. In this example, the subject line gives away important information that the sender probably assumed would be encrypted. But if the subject line is encrypted together with the message body, a mail client must retrieve at least part of the message body to provide a 'folder' view.

The plaintext form of the equivalent DARE Message encoding is:

```
[{
    "annotations":["iAEAiBdGcm9tOiBBbGljZUBleGFtcGxlLmNvbQ",
      "iAEBiBNUbzogYm9iQGV4YW1wbGUuY29t",
      "iAECiClTdWJqZWN0OiBUT1AtU0VDUkVUIFByb2R1Y3QgTGF1bmNoIFRvZG
  F5IQ"
      ],
    "ContentMetaData":"ewogICJjdHkiOiAiYXBwbGljYXRpb24vZXhhbXBsZS
  1tYWlsIn0"},
  "VGhlIENFTyB0b2xkIG1lIHRoZSBwcm9kdWN0IGxhdW5jaCBpcyB0b2RheS4gVG
  VsbCBuby1vbmUh"
  ]
```

This contains the same information as before but the data we might wish to encrypt to protect the confidentiality of the payload is separated from data required for processing.

3.4. Encryption and Integrity

Encryption and integrity protections **MAY** be applied to any DARE Envelope Payload and Annotations.

The following is an encrypted version of the message shown earlier. The payload and annotations have both increased in size as a result of the block cipher padding. The header now includes Recipients and Salt fields to enable the content to be decoded.


```
[{
    "enc":"A256CBC",
    "kid":"EBQI-2VQL-DTRO-FY7I-4EGX-3EIF-CMGF",
    "Salt":"jhJra7u5N1xG4rX2_DycsQ",
    "annotations":["iAEAiCDOkq0Bul3Rb-GFS2g36E3o1E3B3puMrUslxJYIqF
  22rw",
      "iAEBiCANcMr3xLuCsdBS9Phi495YUnurUR7KnOlTYmVLnIxNzQ",
      "iAECiDDGe7i8LCuPatoU2VatkG4RW1S2DorDPWfDqhmGjBP_Tvn0IVN1pk
  6cskOdz3BUajs"
      ],
    "recipients":[{
        "kid":"MDBD-G2N6-CEIS-LQJ5-NNTO-PAHD-OJKW",
        "epk":{
          "PublicKeyECDH":{
            "crv":"Ed25519",
            "Public":"XxF6M_hpG3NCWA6MG9ZDIQNeHvY2ZCZkd7k2j63BtgE"}},
        "wmk":"WNb5zwCz7RTJVB1nzlqdrElUNAdYMCw-wyZNx3Jsl3s5oxJ537
  2h0w"}
      ],
    "ContentMetaData":"ewogICJjdHkiOiAiYXBwbGljYXRpb24vZXhhbXBsZS
  1tYWlsIn0"},
  "alNqzXk6MQmTxQEuDRkhFzo7aij2msYZZSvz--yuOllMWD9y6DosokU1b3zkJY
  VKTbmyU1TZZ9FYQRaqqryIaw"
  ]
```


For efficiency of processing, the ContentMetaData is presented in plaintext. This header could be encrypted as an EDS sequence and presented as a cloaked header.

3.4.1. Key Exchange

The DARE key exchange is based on the JWE key exchange except that encryption modes are intentionally limited and the output of the key exchange is the DARE Master Key rather than the Content Encryption Key.

A DARE Key Exchange **MAY** contain any number of Recipient entries, each providing a means of decrypting the Master Key using a different private key.

If the Key Exchange mechanism supports message recovery, Direct Key Agreement is used, in all other cases, Key Wrapping is used.

This approach allows envelopes with one intended recipient to be handled in the exact same fashion as envelopes with multiple recipients. While this does require an additional key wrapping operation, that could be avoided if an envelope has exactly one intended recipient, this is offset by the reduction in code complexity.

If the key exchange algorithm does not support message recovery (e.g. Diffie Hellman and Elliptic Curve Diffie-Hellman), the HKDF Extract-and-Expand Key Derivation Function is used to derive a master key using the following info tag:

"dare-master" [64 61 72 65 2d 6d 61 73 74 65 72] Key derivation info field used when deriving a master key from the output of a key exchange.

The master key length is the maximum of the key size of the encryption algorithm specified by the key exchange header, the key size of the MAC algorithm specified by the key exchange header (if used) and 256.

3.4.2. Key Identifiers

The JWE/JWS specifications define a kid field for use as a key identifier but not how the identifier itself is constructed. All DARE key identifiers are either UDF key fingerprints [draft-hallambaker-mesh-udf] or Mesh/Recrypt Group Key Identifiers.

A UDF fingerprint is formed as the digest of an IANA content type and the digested data. A UDF key fingerprint is formed with the content type application/pkix-keyinfo and the digested data is the ASN.1 DER encoded PKIX certificate keyInfo sequence for the corresponding public key.

A Group Key Identifier has the form <fingerprint>@<domain>. Where <fingerprint> is a UDF key fingerprint and <domain> is the DNS address of a service that provides the encryption service to support decryption by group members.

3.4.3. Salt Derivation

A Master Salt is a sequence of 16 or more octets that is specified in the Salt field of the header.

The Master Salt is used to derive salt values for the envelope payload and associated encoded data sequences as follows.

Payload Salt = Master Salt

EDS Salt = Concatenate (Payload Salt Prefix, Master Salt)

Encoders **SHOULD NOT** generate salt values that exceed 1024 octets.

The salt value is opaque to the DARE encoding but **MAY** be used to encode application specific semantics including:

- Frame number to allow reassembly of a data sequence split over a sequence of envelopes which may be delivered out of order.
- Transmit the Master Key in the manner of a Kerberos ticket.
- Identify the Master Key under which the Enhanced Data Sequence was generated.
- Enable access to the plaintext to be eliminated by erasure of the encryption key.

For data erasure to be effective, the salt **MUST** be constructed so that the difficulty of recovering the key is sufficiently high that it is infeasible. For most purposes, a salt with 128 bits of appropriately random data is sufficient.

3.4.4. Key Derivation

Encryption and/or authentication keys are derived from the Master Key using a Extract-and-Expand Key Derivation Function as follows:

0. The Master Key and salt value are used to extract the PRK (pseudorandom key)
1. The PRK is used to derive the algorithm keys using the application specific information input for that key type.

The application specific information inputs are:

"dare-encrypt" [64 61 72 65 2d 65 6e 63 72 79 70 74] To generate an encryption or encryption with authentication key.

"dare-iv" [64 61 72 65 2d 65 6e 63 72 79 70 74] To generate an initialization vector.

"dare-mac" [dare-mac] To generate a Message Authentication Code key.

3.5. Signature

While encryption and integrity enhancements can be applied to any part of a DARE Envelope, signatures are only applied to payload digest values calculated over one or more envelope payloads.

The payload digest value for an envelope is calculated over the binary payload data. That is, after any encryption enhancement has been applied but before the envelope encoding is applied. This allows envelopes to be converted from one encoding to another without affecting signature verification.

Single Payload The signed value is the payload digest of the envelope payload.

Multiple Payload. The signed value is the root of a Merkle Tree in which the payload digest of the envelope is one of the leaves.

Verification of a multiple payload signature naturally requires the additional digest values required to construct the Merkle Tree. These are provided in the Trailer in a format that permits multiple signers to reference the same tree data.

3.6. Algorithms

3.6.1. Field: kwd

The key wrapping and derivation algorithms.

Since the means of public key exchange is determined by the key identifier of the recipient key, it is only necessary to specify the algorithms used for key wrapping and derivation.

The default (and so far only) algorithm is kwd-aes-sha2-256-256.

Advanced Encryption Standard (AES) Key Wrap with Padding Algorithm [RFC3394] is used to wrap the Master Exchange Key. AES 256 is used.

HMAC-based Extract-and-Expand Key Derivation Function [RFC5869] is used for key derivation. SHA-2-256 is used for the hash function.

4. DARE Sequence Architecture

4.1. Sequence Navigation

Three means of locating frames in a container are supported:

Sequential Access frames sequentially starting from the start or the end of the container.

Binary search
: Access any container frame by frame number in $O(log_2(n))$ *time by means of a binary tree constructed while the container is written.*

Index
: Access and container frame by frame number or by key by means of an index record.

All DARE Sequences support sequential access. Only tree and Merkle tree containers support binary search access. An index frame **MAY** be written appended to any container and provides $O(1)$ access to any frame listed in the index.

Two modes of compilation are considered:

Monolithic
: Frames are added to the container in a single operation, e.g. file archives,

Incremental
: Additional frames are written to the container at various intervals after it was originally created, e.g. server logs, message spools.

In the monolithic mode, navigation requirements are best met by writing an index frame to the end of the container when it is complete. It is not necessary to construct a binary search tree unless a Merkle tree integrity check is required.

In the incremental mode, Binary search provides an efficient means of locating frames by frame number but not by key. Writing a complete index to the container every m write operations provides $O(m)$ search access but requires $O(n^2)$ *storage.*

Use of partial indexes provides a better compromise between speed and efficiency. A partial index is written out every m frames where m is a power of two. A complete index is written every time a binary tree apex record is written. This approach provides for $O(log_2(n))$ *search with incremental compilation with approximately double the overhead of the monolithic case.*

4.1.1. Tree

As previously described, the JBCD frame structure allows incremental navigation to the immediately preceding frame. The TreePosition parameter specifies the start position of *any* previous frame in the container, thus allowing rapid navigation to that point.

The TreePosition parameter **MAY** be used to enable any frame in the container to be retrieved in $log_2(n)$ time by means of a binary search. The TreePosition parameter specifies the immediately preceding apex of a binary tree formed from the container entries.

For example, the TreePosition of frame 6 in a container gives the location of frame 5, the TreePosition of frame 5 gives the location of frame 3, the TreePosition of frame 3 gives the location of frame 1, and the TreePosition of frame 1 gives the location of frame 0:

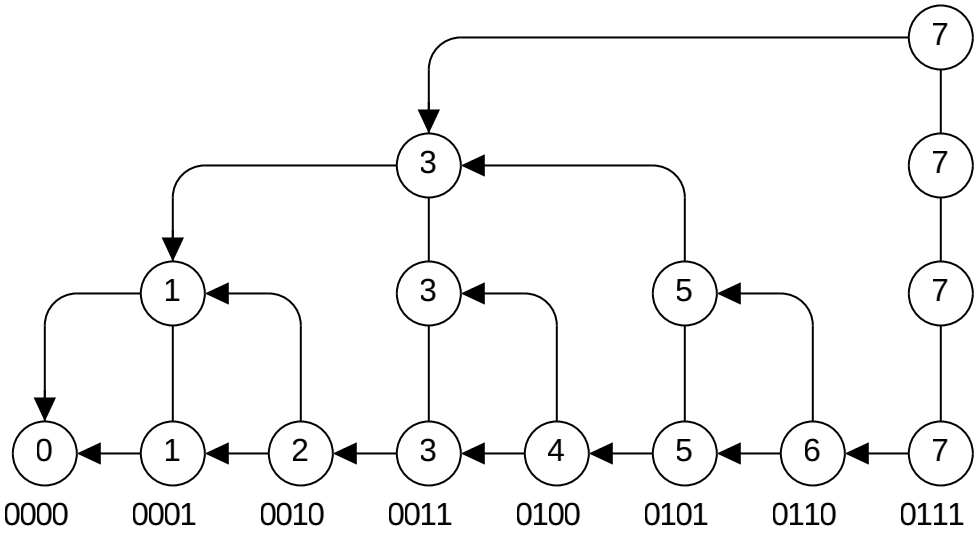

Figure 5: Binary search tree.

An algorithm for efficiently calculating the immediately preceding apex is provided in Appendix C.

4.1.2. Position Index

Contains a table of frame number, position pairs pointing to prior locations in the file.

4.1.3. Metadata Index

Contains a list of IndexMeta entries. Each entry contains a metadata description and a list of frame indexes (not positions) of frames that match the description.

4.2. Integrity Mechanisms

Frame sequences in a DARE container **MAY** be protected against a frame insertion attack by means of a digest chain, a binary Merkle tree or both.

4.2.1. Digest Chain calculation

A digest chain is simple to implement but can only be verified if the full chain of values is known. Appending a frame to the chain has *O(1)* complexity but verification has *O(n)* complexity:

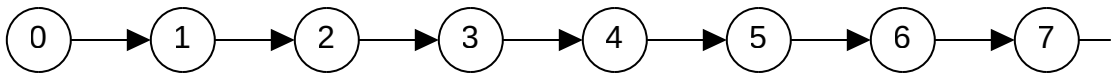

Figure 6: Hash chain integrity check

The value of the chain digest for the first frame (frame 0) is *H(H(null)+H(Payload$_0$)), where null is a zero length octet sequence and payloadn is the sequence of payload data bytes for frame n*

The value of the chain digest for frame *n* is *H(H(Payload$_{n-1}$ + H(Payloadn)), where A+B stands for concatenation of the byte sequences A and B.*

4.2.2. Binary Merkle tree calculation

The tree index mechanism describe earlier may be used to implement a binary Merkle tree. The value TreeDigest specifies the apex value of the tree for that node.

Appending a frame to the chain has *O(log$_2$ (n)) complexity provided that the container format supports at least the binary tree index. Verifying a chain has O(log2 (n)) complexity, provided that the set of necessary digest inputs is known.*

To calculate the value of the tree digest for a node, we first calculate the values of all the sub trees that have their apex at that node and then calculate the digest of that value and the immediately preceding local apex.

4.2.3. Signature

Payload data **MAY** be signed using a JWS [RFC7515] as applied in the Envelope.

Signatures are specified by the Signatures parameter in the content header. The data that the signature is calculated over is defined by the typ parameter of the Signature as follows.

Payload The value of the PayloadDigest parameter

Chain The value of the ChainDigest parameter

Tree The value of the TreeDigestFinal parameter

If the typ parameter is absent, the value Payload is implied.

A frame **MAY** contain multiple signatures created with the same signing key and different typ values.

The use of signatures over chain and tree digest values permit multiple frames to be validated using a single signature verification operation.

5. DARE Schema

A DARE Envelope consists of a Header, an Enhanced Data Sequence (EDS) and an optional trailer. This section describes the JSON data fields used to construct headers, trailers and complete envelopes.

Wherever possible, fields from JWE, JWS and JWK have been used. In these cases, the fields have the exact same semantics. Note however that the classes in which these fields are presented have different structure and nesting.

5.1. Envelope Classes

A DARE envelope contains a single DAREMessageSequence in either the JSON or Compact serialization as directed by the protocol in which it is applied.

5.1.1. Structure: DareEnvelopeSequence

A DARE envelope containing Header, EDS and Trailer in JSON object encoding. Since a DAREMessage is almost invariably presented in JSON sequence or compact encoding, use of the DAREMessage subclass is preferred.

Although a DARE envelope is functionally an object, it is serialized as an ordered sequence. This ensures that the envelope header field will always precede the body in a serialization, this allowing processing of the header information to be performed before the entire body has been received.

Header: DareHeader (Optional) The envelope header. May specify the key exchange data, pre-signature or signature data, cloaked headers and/or encrypted data sequences.

Body: Binary (Optional) The envelope body

Trailer: DareTrailer (Optional) The envelope trailer. If present, this contains the signature.

5.2. Header and Trailer Classes

A DARE sequence MUST contain a (possibly empty) DAREHeader and MAY contain a DARETrailer.

5.2.1. Structure: DareTrailer

A DARE envelope Trailer

Signatures: DareSignature [0..Many] A list of signatures. A envelope trailer MUST NOT contain a signatures field if the header contains a signatures field.

SignedData: Binary (Optional) Contains a DAREHeader object

PayloadDigest: Binary (Optional) If present, contains the digest of the Payload.

ChainDigest: Binary (Optional) If present, contains the digest of the PayloadDigest values of this frame and the frame immediately preceding.

TreeDigest: Binary (Optional) If present, contains the Binary Merkle Tree digest value.

5.2.2. Structure: DareHeader

Inherits: DareTrailer

A DARE Envelope Header. Since any field that is present in a trailer MAY be placed in a header instead, the envelope header inherits from the trailer.

EnvelopeId: String (Optional) Unique identifier

EncryptionAlgorithm: String (Optional) The encryption algorithm as specified in JWE

DigestAlgorithm: String (Optional) Digest Algorithm. If specified, tells decoder that the digest algorithm is used to construct a signature over the envelope payload.

KeyIdentifier: String (Optional) Base seed identifier.

Salt: Binary (Optional) Salt value used to derrive cryptographic parameters for the content data.

Malt: Binary (Optional) Hash of the Salt value used to derrive cryptographic parameters for the content data. This field SHOULD NOT be present if the Salt field is present. It is used to allow the salt value to be erased (thus rendering the payload content irrecoverable) without affecting the ability to calculate the payload digest value.

Cloaked: Binary (Optional) If present in a header or trailer, specifies an encrypted data block containing additional header

fields whose values override those specified in the envelope and context headers.

When specified in a header, a cloaked field MAY be used to conceal metadata (content type, compression) and/or to specify an additional layer of key exchange. That applies to both the envelope body and to headers specified within the cloaked header.

Processing of cloaked data is described in...

EDSS: Binary [0..Many] If present, the Annotations field contains a sequence of Encrypted Data Segments encrypted under the envelope base seed. The interpretation of these fields is application specific.

Recipients: DareRecipient [0..Many] A list of recipient key exchange information blocks.

Policy: DarePolicy (Optional) A DARE security policy governing future additions to the container.

ContentMetaData: Binary (Optional) If present contains a JSON encoded ContentInfo structure which specifies plaintext content metadata and forms one of the inputs to the envelope digest value.

SequenceInfo: SequenceInfo (Optional) Information that describes container information

SequenceIndex: SequenceIndex (Optional) An index of records in the current container up to but not including this one.

Received: DateTime (Optional) Date on which the envelope was received.

Cover: Binary (Optional) HTML document containing cover text to be presented if the document cannot be decrypted.

5.2.3. Structure: ContentMeta

UniqueId: String (Optional) Unique object identifier

Labels: String [0..Many] List of labels that are applied to the payload of the frame.

KeyValues: KeyValue [0..Many] List of key/value pairs describing the payload of the frame.

MessageType: String (Optional) The mesh message type

ContentType: String (Optional)
The content type field as specified in JWE

Paths: String [0..Many] List of filename paths for the payload of the frame.

Filename: String (Optional) The original filename under which the data was stored.

Event: String (Optional) Operation on the header

Created: DateTime (Optional) Initial creation date.

Modified: DateTime (Optional) Date of last modification.

Expire: DateTime (Optional) Date at which the associated transaction will expire

First: Integer (Optional) Frame number of the first object instance value.

Previous: Integer (Optional) Frame number of the immediately prior object instance value

FileEntry: FileEntry (Optional) Information describing the file entry on disk.

5.3. Cryptographic Data

DARE envelope uses the same fields as JWE and JWS but with different structure. In particular, DARE envelopes MAY have multiple recipients and multiple signers.

5.3.1. Structure: DareSignature

The signature value

Dig: String (Optional)
Digest algorithm hint. Specifying the digest algorithm to be applied to the envelope body allows the body to be processed in streaming mode.

Alg: String (Optional) Key exchange algorithm

KeyIdentifier: String (Optional) Key identifier of the signature key.

Certificate: X509Certificate (Optional) PKIX certificate of signer.

Path: X509Certificate (Optional) PKIX certificates that establish a trust path for the signer.

Manifest: Binary (Optional) The data description that was signed.

SignatureValue: Binary (Optional) The signature value as an Enhanced Data Sequence under the envelope base seed.

WitnessValue: Binary (Optional) The signature witness value used on an encrypted envelope to demonstrate that the signature was authorized by a party with actual knowledge of the encryption key used to encrypt the envelope.

5.3.2. Structure: X509Certificate

X5u: String (Optional) URL identifying an X.509 public key certificate

X5: Binary (Optional) An X.509 public key certificate

5.3.3. Structure: DareRecipient

Recipient information

KeyIdentifier: String (Optional) Key identifier for the encryption key.

The Key identifier MUST be either a UDF fingerprint of a key or a Group Key Identifier

KeyWrapDerivation: String (Optional) The key wrapping and derivation algorithms.

WrappedBaseSeed: Binary (Optional) The wrapped base seed. The base seed is encrypted under the result of the key exchange.

RecipientKeyData: String (Optional) The per-recipient key exchange data.

5.3.4. Structure: DarePolicy

EncryptionAlgorithm: String (Optional) The encryption algorithm to be used to compute the payload.

DigestAlgorithm: String (Optional) The digest algorithm to be used to compute the payload digest.

Encryption: String (Optional) The encryption policy specifier, determines how often a key exchange is required. 'Single': All entries are encrypted under the key exchange specified in the entry specifying this policy. 'Isolated': All entries are encrypted under a separate key exchange. 'All': All entries are encrypted. 'None': No entries are encrypted.

Default value is 'None' if EncryptKeys is null, and 'All' otherwise.

Signature: String (Optional) The signature policy 'None': No entries are signed. 'Last': The last entry in the container is signed. 'Isolated': All entries are independently signed. 'Any': Entries may be signed.

Default value is 'None' if SignKeys is null, and 'Any' otherwise.

Sealed: Boolean (Optional) If true the policy is immutable and cannot be changed by a subsequent policy override.

5.3.5. Structure: FileEntry

Path: String (Optional) The file path in canonical form.

CreationTime: DateTime (Optional) The creation time of the file on disk in UTC

LastAccessTime: DateTime (Optional) The last access time of the file on disk in UTC

LastWriteTime: DateTime (Optional) The last write time of the file on disk in UTC

Attributes: Integer (Optional) The file attribues as a bitmapped integer.

5.3.6. Structure: Witness

Entry containing the latest apex value of a specified append only log.

Id: String (Optional) Globally unique log identifier

Issuer: String (Optional)
The issuer of the log

Apex: Binary (Optional) The Apex hash value

Index: Integer (Optional) Specifies the index number assigned to the entry in the log.

5.3.7. Structure: Proof

Provides a proof that the payload with digest [hash] in the log described by SignedWitness occurs at the index [Index]

SignedWitness: DareEnvelope (Optional) The signed apex under which this proof chain is established

Hash: Binary (Optional) Specifies the index number assigned to the
Index: Integer (Optional) entry in the log.

Path: Binary [0..Many] The list of entries from which the proof path is computed.

6. DARE Container Schema

TBS stuff

6.1. Container Headers

TBS stuff

6.1.1. Structure: SequenceInfo

Information that describes container information

DataEncoding: String (Optional) Specifies the data encoding for the header section of for the following frames. This value is ONLY valid in Frame 0 which MUST have a header encoded in JSON.

ContainerType: String (Optional) Specifies the container type for the following records. This value is ONLY valid in Frame 0 which MUST have a header encoded in JSON.

Index: Integer (Optional) The record index within the file. This MUST be unique and satisfy any additional requirements determined by the ContainerType.

IsMeta: Boolean (Optional) If true, the current frame is a meta frame and does not contain a payload.

Note: Meta frames MAY be present in any container. Applications MUST accept containers that contain meta frames at any position in the file. Applications MUST NOT interpret a meta frame as a data frame with an enpty payload.

Default: Boolean (Optional) If set true in a persistent container, specifies that this record contains the default object for the container.

TreePosition: Integer (Optional) Position of the frame containing the apex of the preceding sub-tree.

IndexPosition: Integer (Optional) Specifies the position in the file at which the last index entry is to be found

ExchangePosition: Integer (Optional) Specifies the position in the file at which the key exchange data is to be found

6.2. Index Structures

TBS stuff

6.2.1. Structure: SequenceIndex

A container index

Full: Boolean (Optional) If true, the index is complete and contains position entries for all the frames in the file. If absent or false, the index is incremental and only contains position entries for records added since the last frame containing a ContainerIndex.

Positions: IndexPosition [0..Many] List of container position entries

6.2.2. Structure: IndexPosition

Specifies the position in a file at which a specified record index is found

Index: Integer (Optional) The record index within the file.

Position: Integer (Optional) The record position within the file relative to the index base.

UniqueId: String (Optional) Unique object identifier

6.2.3. Structure: KeyValue

Specifies a key/value entry

Key: String (Optional)
The key

Value: String (Optional) The value corresponding to the key

7. Dare Sequence Applications

DARE Sequences are used to implement two forms of persistence store to support Mesh operations:

Catalogs A set of related items which **MAY** be added, modified or deleted at any time.

Spools A list of related items whose status **MAY** be changed at any time but which are immutable once added.

Since DARE Sequences are an append only log format, entries can only be modified or deleted by adding items to the log to change the status of previous entries. It is always possible to undo any operation on a catalog or spool unless the underlying container is purged or the individual entries modified.

7.1. Catalog

Catalogs contain a set of entries, each of which is distinguished by a unique identifier.

Three operations are supported:

Add Addition of the entry to the catalog

Update Modification of the data associated with the entry excluding the identifier

Delete Removal of the entry from the catalog

The set of valid state transitions is defined by the Finite State machine:

(Add-Update*-Delete)*

Catalogs are used to represent sets of persistent objects associated with a Mesh Service Account. The user's set of contacts for example. Each contact entry may be modified many times over time but refers to the same subject for its entire lifetime.

7.2. Spool

Spools contain lists of entries, each of which is distinguished by a unique identifier.

Four operations are supported:

Post Addition of the entry to the spool

Processed Marks the entry as having been processed.

Unprocessed Returns the entry to the unread state.

Delete Mark the entry as deleted allowing recovery of associated storage in a subsequent purge operation.

The set of valid state transitions is defined by the Finite State machine:

Post-(Processed| Unprocessed| Delete *)

Spools are used to represent time sequence ordered entries such as lists of messages being sent or received, task queues and transaction logs.

7.3. Archive

A DARE Archive is a DARE Sequence whose entries contain files. This affords the same functionality as a traditional ZIP or tar archive but with the added cryptographic capabilities provided by the DARE format.

8. Future Work

The current specification describes an approach in which containers are written according to a strict append-only policy. Greater flexibility may be achieved by loosening this requirement allowing record(s) at the end of the container to be overwritten.

8.1. Terminal integrity check

A major concern when operating a critical service is the possibility of a hardware or power failure occurring during a write operation causing the file update to be incomplete. While most modern operating systems have effective mechanisms in place to prevent corruption of the file system itself in such circumstances, this does not provide sufficient protection at the application level.

Appending a null record containing a container-specific magic number provides an effective means of detecting this circumstance that can be quickly verified.

If a container specifies a terminal integrity check value in the header of frame zero, the container is considered to be in an

incomplete write state if the final frame is not a null record specifying the magic number.

When appending new records to such containers, the old terminal integrity check record is overwritten by the data being added and a new integrity check record appended to the end.

8.2. Terminal index record

A writer can maintain a complete (or partial) index of the container in its final record without additional space overhead by overwriting the prior index on each update.

8.3. Deferred indexing

The task of updating terminal indexes may be deferred to a time when the machine is not busy. This improves responsiveness and may avoid the need to re-index containers receiving a sequence of updates.

This approach may be supported by appending new entries to the end of the container in the usual fashion and maintaining a record of containers to be updated as a separate task.

When updating the index on a container that has been updated in this fashion, the writer must ensure that no data is lost even if the process is interrupted. The use of guard records and other precautions against loss of state is advised.

9. Security Considerations

This section describes security considerations arising from the use of DARE in general applications.

Additional security considerations for use of DARE in Mesh services and applications are described in the Mesh Security Considerations guide [draft-hallambaker-mesh-security].

9.1. Encryption/Signature nesting

9.2. Side channel

9.3. Salt reuse

10. IANA Considerations

11. Acknowledgements

A list of people who have contributed to the design of the Mesh is presented in [draft-hallambaker-mesh-architecture].

The name Data At Rest Encryption was proposed by Melhi Abdulhayo?lu.

12. Appendix A: DARE Envelope Examples and Test Vectors

13. Test Examples

In the following examples, Alice's encryption private key parameters are:

```
{
  "PrivateKeyECDH":{
    "Private":"ZTpkcKy-6Q_meq-jbtUJNjVCcC0G7udaagPGKtYywjY",
    "crv":"Ed25519"}}
```

Alice's signature private key parameters are:

```
{
  "PrivateKeyECDH":{
    "Private":"zNBnGilzHnOQ2CpoQGZBNM7Z5FmqvEqQaPl3_hNJIJ4",
    "crv":"Ed25519"}}
```

The body of the test message is the UTF8 representation of the following string:

```
"This is a test long enough to require multiple blocks"
```

The EDS sequences, are the UTF8 representation of the following strings:

```
"Subject: Message metadata should be encrypted"
"2018-02-01"
```

13.1. Plaintext Message

A plaintext message without associated EDS sequences is an empty header followed by the message body:

```
{
  "DareEnvelope":[{},
    "VGhpcyBpcyBhIHRlc3QgbG9uZyBlbm91Z2ggdG8gcmVxdWlyZSBtdWx0aXBs
  ZSBibG9ja3M"
    ]}
```

13.2. Plaintext Message with EDS

If a plaintext message contains EDS sequences, these are also in plaintext:

```
{
  "DareEnvelope":[{
      "annotations":["iAEAiC1TdWJqZWN0OiBNZXNzYWdlIG1ldGFkYXRhIHNo
  b3VsZCBiZSBlbmNyeXB0ZWQ",
        "iAEBiAoyMDE4LTAyLTAx"
        ]},
    "VGhpcyBpcyBhIHRlc3QgbG9uZyBlbm91Z2ggdG8gcmVxdWlyZSBtdWx0aXBs
  ZSBibG9ja3M"
    ]}
```

13.3. Encrypted Message

The creator generates a base seed:

```
69 CC E0 59  36 2C A3 AC  B5 BB F3 6E  02 C6 2F 9C
62 C8 F9 01  25 F9 BF 95  27 45 50 3E  34 EB 3C 4E
```

For each recipient of the message:

The creator generates an ephemeral key:

```
{
  "PrivateKeyECDH":{
    "Private":"ya_wfOr6qUsC-SK007UBc4_N-rgJOA938gHYAtwmYK8",
    "crv":"Ed25519"}}
```

The key agreement value is calculated:

```
E8 E9 98 18  3D 76 7A AC  40 F6 A9 17  59 9D A3 E5
6F 3D 33 C8  5C 72 3B A5  0C 1C 0B E5  77 C0 BC 60
```

The key agreement value is used as the input to a HKDF key derivation function with the info parameter master to create the key used to wrap the base seed:

```
E6 05 E7 B0  CA FC 9F 71  6D D5 D2 51  EF EE AE 13
7D 0E C3 41  3C 43 6E 1B  F4 C3 DB D1  F0 74 90 3C
```

The wrapped base seed is:

```
58 D6 F9 CF  00 B3 ED 14  C9 54 1D 67  CE 5A 9D AC
49 54 34 07  58 30 2C 3E  C3 26 4D C7  72 6C 97 7B
39 A3 12 79  DF BD A1 D3
```

This information is used to calculate the Recipient information shown in the example below.

To encrypt a message, we first generate a unique salt value:

```
5A 14 91 3A  02 37 54 28  EF 3F C5 CD  19 75 40 A8
```

The base seed and salt value are used to generate the payload encryption key:

```
AA 25 C0 99  EE BA 06 60  82 A1 5E 1D  F3 BF 8E BA
20 99 94 3A  92 6B 4D 97  6B B9 3B 25  7C 05 F0 CB
```

Since AES is a block cipher, we also require an initializarion vector:

```
48 D3 64 35  CB 41 A6 6E  9D 5E 4D 11  85 C0 B3 7B
```

The output sequence is the encrypted bytes:

```
20 88 B5 BD  F3 50 6F 35  1E 02 E0 50  96 8B 7E 0D
FB 42 42 67  EB E9 C4 86  0E 79 80 A8  B5 34 4F 66
31 8A 99 67  BD 4C 14 92  46 6D FD 68  FF 6E 1A DD
3F D6 D1 41  DF 61 DD 0D  6F DB E9 84  9C 96 3E A6
```

Since the message is not signed, there is no need for a trailer. The completed message is:

```
{
  "DareEnvelope":[{
      "enc":"A256CBC",
      "kid":"EBQF-ULB3-QRUJ-DJ65-VW6U-DCMS-7IUW",
      "Salt":"WhSROgI3VCjvP8XNGXVAqA",
      "recipients":[{
          "kid":"MDBD-G2N6-CEIS-LQJ5-NNTO-PAHD-OJKW",
          "epk":{
            "PublicKeyECDH":{
              "crv":"Ed25519",
              "Public":"9eX_-YhN7TaNYxTLR-4m1D424Bb5Oee1yDiYauwvR
  uM"}},
          "wmk":"rVnqGDdpDtIr0vlNtOFR1XZsHgt-xCsmnct7CBTx8dfcd9CI
  N6-IpQ"}
        ]},
    "IIi1vfNQbzUeAuBQlot-DftCQmfr6cSGDnmAqLU0T2YxiplnvUwUkkZt_Wj_
  bhrdP9bRQd9h3Q1v2-mEnJY-pg"
    ]}
```

13.4. Signed Message

Signed messages specify the digest algorithm to be used in the header and the signature value in the trailer. Note that the digest algorithm is not optional since it serves as notice that a decoder should digest the payload value to enable signature verification.

```
{
  "DareEnvelope":[{
      "dig":"S512"},
    "VGhpcyBpcyBhIHRlc3QgbG9uZyBlbm91Z2ggdG8gcmVxdWlyZSBtdWx0aXBs
  ZSBibG9ja3M",
    {
      "signatures":[{
          "alg":"S512",
          "kid":"MBNF-MADB-OAZ3-7JOJ-WCQA-LQEK-JZTT",
          "signature":"HjotuHPSvOCJ8-qmctwpwfr1oTBlKV_zS9SwC3Rewd
  tpfVDOYMbfcY65ppf08lc5a158zNzohwpecH09_RUnBg"}
        ],
      "PayloadDigest":"raim8SV5adPbWWn8FMM4mrRAQCO9A2jZ0NZAnFXWlG
  0xF6sWGJbnKSdtIJMmMU_hjarlIPEoY3vy9UdVlH5KAg"}
    ]}
```

13.5. Signed and Encrypted Message

A signed and encrypted message is encrypted and then signed. The signer proves knowledge of the payload plaintext by providing the plaintext witness value.


```
{
  "DareEnvelope":[{
      "enc":"A256CBC",
      "dig":"S512",
      "kid":"EBQE-JDZA-O6SC-THAF-SY5Q-27HI-N7M6",
      "Salt":"Q0WHe4ToWVPt8B_yZMsTew",
      "recipients":[{
          "kid":"MDBD-G2N6-CEIS-LQJ5-NNTO-PAHD-OJKW",
          "epk":{
            "PublicKeyECDH":{
              "crv":"Ed25519",
              "Public":"wi5FlXOMcCjzMtB-zo2TBjCUKeqMvAFs3JWsrxtbd
  fw"}},
          "wmk":"tScSFm_ixIAFV4uGXiwbflJOlEmI2vEocGXh7DWgOYFcl4Br
  _DTzkg"}
        ]},
    "urNDz3AmiSBuVbkXE-AltI8m9Xkt0q1Ar24hlcrNrCQ21xiykC_iz39p1jKz
  KWNJssLgR8NWyTSzzkZ00FXlvQ",
    {
      "signatures":[{
          "alg":"S512",
          "kid":"MBNF-MADB-OAZ3-7JOJ-WCQA-LQEK-JZTT",
          "signature":"54QC4ridASGMyMlBj7b9iIruU0RuF3QBZYbHvsfRdj
  WrDkoKWRFK1LDKYJ1b0asv8G_zuR4IoLJbSS_ayVTBAg",
          "witness":"OPjhVd75ROFKAMzXWS70zyNqMN5PD30xb1kaSzZ_u5E"}
        ],
      "PayloadDigest":"RVr3LlUYe-19Q3HIQeUQAjqXYZ9b0V7ehdxBNm9tU6
  SxQEYWykL_eq-h4qu5xTbNUdfvtMS6mEcbIJpFG19rtg"}
    ]}
```


14. Appendix B: DARE Sequence Examples and Test Vectors

The data payloads in all the following examples are identical, only the authentication and/or encryption is different.

*Frame 1..n consists of 300 bytes being the byte sequence 00, 01, 02, etc. repeating after 256 bytes.

For conciseness, the raw data format is omitted for examples after the first, except where the data payload has been transformed, (i.e. encrypted).

14.1. Simple sequence

The following example shows a simple sequence with first frame and a single data frame:

```
f4 91
f0 8b
f0 00
f0 00
91 f4
f5 01 73
f0 42
f1 01 2c
73 01 f5
```

Since there is no integrity check, there is no need for trailer entries. The header values are:

Frame 0

```
{
  "DareHeader":{
    "policy":{},
    "ContentMetaData":"e30",
    "SequenceInfo":{
      "DataEncoding":"JSON",
      "ContainerType":"List",
      "Index":0}}}

[Empty trailer]
```

Frame 1

```
{
  "DareHeader":{
    "ContentMetaData":"e30",
    "SequenceInfo":{
      "Index":1}}}

[Empty trailer]
```

14.2. Payload and chain digests

The following example shows a chain sequence with a first frame and three data frames. The headers of these frames is the same as before but the frames now have trailers specifying the PayloadDigest and ChainDigest values:

Frame 0

```
{
  "DareHeader":{
    "dig":"S512",
    "policy":{},
    "ContentMetaData":"e30",
    "SequenceInfo":{
      "DataEncoding":"JSON",
      "ContainerType":"Chain",
      "Index":0}}}

[Empty trailer]
```

Frame 1

```
{
  "DareHeader":{
    "dig":"S512",
    "ContentMetaData":"e30",
    "SequenceInfo":{
      "Index":1}}}

{
  "DareTrailer":{
    "PayloadDigest":"8dyi62d7MDJlsLm6_w4GEgKBjzXBRwppu6qbtmAl6UjZ
  DlZeaWQlBsYhOu88-ekpNXpZ2iY96zTRI229zaJ5sw",
    "ChainDigest":"T7S1FcrgY3AaWD4L-t5W1K-3XYkPTcOdGEGyjglTD6yMYV
  RVz9tn_KQc6GdA-P4VSRigBygV65OEd2Vv3YDhww"}}
```

Frame 2

```
{
  "DareHeader":{
    "dig":"S512",
    "ContentMetaData":"e30",
    "SequenceInfo":{
      "Index":2}}}

{
  "DareTrailer":{
    "PayloadDigest":"8dyi62d7MDJlsLm6_w4GEgKBjzXBRwppu6qbtmAl6UjZ
  DlZeaWQlBsYhOu88-ekpNXpZ2iY96zTRI229zaJ5sw",
    "ChainDigest":"T7S1FcrgY3AaWD4L-t5W1K-3XYkPTcOdGEGyjglTD6yMYV
  RVz9tn_KQc6GdA-P4VSRigBygV65OEd2Vv3YDhww"}}
```

Frame 3

```
{
  "DareHeader":{
    "dig":"S512",
    "ContentMetaData":"e30",
    "SequenceInfo":{
      "Index":3}}}

{
  "DareTrailer":{
    "PayloadDigest":"8dyi62d7MDJlsLm6_w4GEgKBjzXBRwppu6qbtmAl6UjZ
  DlZeaWQlBsYhOu88-ekpNXpZ2iY96zTRI229zaJ5sw",
    "ChainDigest":"T7S1FcrgY3AaWD4L-t5W1K-3XYkPTcOdGEGyjglTD6yMYV
  RVz9tn_KQc6GdA-P4VSRigBygV65OEd2Vv3YDhww"}}
```

14.3. Merkle Tree

The following example shows a chain sequence with a first frame and six data frames. The trailers now contain the TreePosition and TreeDigest values:

Frame 0

```
{
  "DareHeader":{
    "dig":"S512",
    "policy":{},
    "ContentMetaData":"e30",
    "SequenceInfo":{
      "DataEncoding":"JSON",
      "ContainerType":"Merkle",
      "Index":0}}}

[Empty trailer]
```


Frame 1


```
{
  "DareHeader":{
    "dig":"S512",
    "ContentMetaData":"e30",
    "SequenceInfo":{
      "Index":1,
      "TreePosition":0}}}

{
  "DareTrailer":{
    "PayloadDigest":"8dyi62d7MDJlsLm6_w4GEgKBjzXBRwppu6qbtmAl6UjZ
  DlZeaWQlBsYhOu88-ekpNXpZ2iY96zTRI229zaJ5sw",
    "TreeDigest":"T7S1FcrgY3AaWD4L-t5W1K-3XYkPTcOdGEGyjglTD6yMYVR
  Vz9tn_KQc6GdA-P4VSRigBygV65OEd2Vv3YDhww"}}
```


Frame 2


```
{
  "DareHeader":{
    "dig":"S512",
    "ContentMetaData":"e30",
    "SequenceInfo":{
      "Index":2,
      "TreePosition":392}}}

{
  "DareTrailer":{
    "PayloadDigest":"8dyi62d7MDJlsLm6_w4GEgKBjzXBRwppu6qbtmAl6UjZ
  DlZeaWQlBsYhOu88-ekpNXpZ2iY96zTRI229zaJ5sw",
    "TreeDigest":"7fHmkEIsPkN6sDYAOLvpIJn5Dg3PxDDAaq-ll2kh8722kok
  kFnZQcYtjuVC71aHNXI18q-lPnfRkmwryG-bhqQ"}}
```

Frame 3


```
{
  "DareHeader":{
    "dig":"S512",
    "ContentMetaData":"e30",
    "SequenceInfo":{
      "Index":3,
      "TreePosition":392}}}

{
  "DareTrailer":{
    "PayloadDigest":"8dyi62d7MDJlsLm6_w4GEgKBjzXBRwppu6qbtmAl6UjZ
  DlZeaWQlBsYhOu88-ekpNXpZ2iY96zTRI229zaJ5sw",
    "TreeDigest":"T7S1FcrgY3AaWD4L-t5W1K-3XYkPTcOdGEGyjglTD6yMYVR
  Vz9tn_KQc6GdA-P4VSRigBygV65OEd2Vv3YDhww"}}
```


Frame 4


```
{
  "DareHeader":{
    "dig":"S512",
    "ContentMetaData":"e30",
    "SequenceInfo":{
      "Index":4,
      "TreePosition":1676}}}

{
  "DareTrailer":{
    "PayloadDigest":"8dyi62d7MDJlsLm6_w4GEgKBjzXBRwppu6qbtmAl6UjZ
  DlZeaWQlBsYhOu88-ekpNXpZ2iY96zTRI229zaJ5sw",
    "TreeDigest":"vJ6ngNATvZcXSMALi5IUqzl1GBxBnTNVcC87VL_BhMRCbAv
  KSj8gs0VFgxxLkZ2myrtaDIwhHoswiTiBMLNWug"}}
```


Frame 5

```
{
  "DareHeader":{
    "dig":"S512",
    "ContentMetaData":"e30",
    "SequenceInfo":{
      "Index":5,
      "TreePosition":1676}}}

{
  "DareTrailer":{
    "PayloadDigest":"8dyi62d7MDJlsLm6_w4GEgKBjzXBRwppu6qbtmAl6UjZ
  DlZeaWQlBsYhOu88-ekpNXpZ2iY96zTRI229zaJ5sw",
    "TreeDigest":"T7S1FcrgY3AaWD4L-t5W1K-3XYkPTcOdGEGyjglTD6yMYVR
  Vz9tn_KQc6GdA-P4VSRigBygV65OEd2Vv3YDhww"}}
```

Frame 6

```
{
  "DareHeader":{
    "dig":"S512",
    "ContentMetaData":"e30",
    "SequenceInfo":{
      "Index":6,
      "TreePosition":2963}}}

{
  "DareTrailer":{
    "PayloadDigest":"8dyi62d7MDJlsLm6_w4GEgKBjzXBRwppu6qbtmAl6UjZ
  DlZeaWQlBsYhOu88-ekpNXpZ2iY96zTRI229zaJ5sw",
    "TreeDigest":"WgHlz3EHczVPqgtpc39Arv7CFIsCbFVsk8wg0j2qLlEfur9
  SZ0mdr65Ka-HF0Qx8gg_DAoiJwUrwADDXyxVJOg"}}
```

14.4. Signed sequence

The following example shows a tree sequence with a signature in the final record. The signing key parameters are:

```
{
  "PrivateKeyECDH":{
    "Private":"zNBnGilzHnOQ2CpoQGZBNM7Z5FmqvEqQaPl3_hNJIJ4",
    "crv":"Ed25519"}}
```

The sequence headers and trailers are:

Frame 0


```
{
  "DareHeader":{
    "dig":"S512",
    "policy":{},
    "ContentMetaData":"e30",
    "SequenceInfo":{
      "DataEncoding":"JSON",
      "ContainerType":"Merkle",
      "Index":0}}}
```


[Empty trailer]

Frame 1


```
{
  "DareHeader":{
    "dig":"S512",
    "ContentMetaData":"e30",
    "SequenceInfo":{
      "Index":1,
      "TreePosition":0}}}

{
  "DareTrailer":{
    "PayloadDigest":"8dyi62d7MDJlsLm6_w4GEgKBjzXBRwppu6qbtmAl6UjZ
  DlZeaWQlBsYhOu88-ekpNXpZ2iY96zTRI229zaJ5sw",
    "TreeDigest":"T7S1FcrgY3AaWD4L-t5W1K-3XYkPTcOdGEGyjglTD6yMYVR
  Vz9tn_KQc6GdA-P4VSRigBygV65OEd2Vv3YDhww"}}
```


Frame 2

```
{
  "DareHeader":{
    "dig":"S512",
    "ContentMetaData":"e30",
    "SequenceInfo":{
      "Index":2,
      "TreePosition":392}}}

{
  "DareTrailer":{
    "PayloadDigest":"8dyi62d7MDJlsLm6_w4GEgKBjzXBRwppu6qbtmAl6UjZ
  DlZeaWQlBsYhOu88-ekpNXpZ2iY96zTRI229zaJ5sw",
    "TreeDigest":"7fHmkEIsPkN6sDYAOLvpIJn5Dg3PxDDAaq-ll2kh8722kok
  kFnZQcYtjuVC71aHNXI18q-lPnfRkmwryG-bhqQ"}}
```

14.5. Encrypted sequence

The following example shows a sequence in which all the frame payloads are encrypted under the same base seed established in a key agreement specified in the first frame.

Frame 0

```
{
  "DareHeader":{
    "enc":"A256CBC",
    "kid":"EBQJ-M6BL-NPCW-YVAB-N5P3-SGMU-JWSM",
    "Salt":"GK35Flr-AkIDXH0F_8eaSw",
    "recipients":[{
        "kid":"MDBD-G2N6-CEIS-LQJ5-NNTO-PAHD-OJKW",
        "epk":{
          "PublicKeyECDH":{
            "crv":"Ed25519",
            "Public":"OMwPhLRENxSIvI5ntJG_B5iQBr4f2Mg1d4M2TfkNCxg"}},
        "wmk":"hVFbhlDNBhNE4KIJ6eGvfN57eP1-3RR434H3kNnr7LvI8XHl-b
  JvTQ"}
      ],
    "policy":{
      "enc":"none",
      "dig":"none",
      "EncryptKeys":[{
          "PublicKeyECDH":{
            "crv":"Ed25519",
            "Public":"6LaCG5duVocIyW7IAtqJoKiGEdAEwQ0xyJs8v7QhH8I"}}
        ],
      "Sealed":true},
    "ContentMetaData":"e30",
    "SequenceInfo":{
      "DataEncoding":"JSON",
      "ContainerType":"List",
      "Index":0}}}

[Empty trailer]
```


Frame 1

```
{
  "DareHeader":{
    "enc":"A256CBC",
    "kid":"EBQO-7MFX-H2VS-DZ5P-PHFS-BVVI-7CGD",
    "Salt":"WMWmVh2-nHezL7LMdj-DUw",
    "recipients":[{
        "kid":"MDBD-G2N6-CEIS-LQJ5-NNTO-PAHD-OJKW",
        "epk":{
          "PublicKeyECDH":{
            "crv":"Ed25519",
            "Public":"vpvcqq0g7LIftivCVRYd2KvznaVmn_4OYg2EKrvoYHI"}},
        "wmk":"xRsKdXMtqGXBDz3ifux_0Tw3i5HiuZUK7x2jrEKijhDA6PbqcN
  IJRA"}
      ],
    "ContentMetaData":"e30",
    "SequenceInfo":{
      "Index":1}}}

[Empty trailer]
```

Frame 2

```
{
  "DareHeader":{
    "enc":"A256CBC",
    "kid":"EBQN-HQYA-I35A-ZLTG-BLAV-3NWG-FLIJ",
    "Salt":"N9IVoifEEGfA67eHuMYQGg",
    "recipients":[{
        "kid":"MDBD-G2N6-CEIS-LQJ5-NNTO-PAHD-OJKW",
        "epk":{
          "PublicKeyECDH":{
            "crv":"Ed25519",
            "Public":"GXNTgfmDDCGDJtiUEDMPtWBhmXKEMjHSc73f7_OAzu8"}},
        "wmk":"puogbB8RpS5klIFemyjNOSy0olhKGB_gJZg7uHtHVVpVEj8f3b
  tebQ"}
      ],
    "ContentMetaData":"e30",
    "SequenceInfo":{
      "Index":2}}}

[Empty trailer]
```

Here are the sequence bytes. Note that the content is now encrypted and has expanded by 25 bytes. These are the salt (16 bytes), the AES padding (4 bytes) and the JSON-B framing (5 bytes).

```
f5 02 ef
f1 02 d8
f0 10
f0 00
ef 02 f5
f5 02 f9
f1 01 c3
f1 01 30
f9 02 f5
f5 02 f9
f1 01 c3
f1 01 30
f9 02 f5
```

The following example shows a sequence in which all the frame payloads are encrypted under separate key agreements specified in the payload frames.

Frame 0

```
{
  "DareHeader":{
    "policy":{
      "enc":"none",
      "dig":"none",
      "Sealed":true},
    "ContentMetaData":"e30",
    "SequenceInfo":{
      "DataEncoding":"JSON",
      "ContainerType":"List",
      "Index":0}}}

[Empty trailer]
```

Frame 1

```
{
  "DareHeader":{
    "ContentMetaData":"e30",
    "SequenceInfo":{
      "Index":1}}}

[Empty trailer]
```

Frame 2

```
{
  "DareHeader":{
    "ContentMetaData":"e30",
    "SequenceInfo":{
      "Index":2}}}

[Empty trailer]
```

15. Appendix C: Previous Frame Function

```
public long PreviousFrame (long Frame) {
    long x2 = Frame + 1;
    long d = 1;

    while (x2 > 0) {
        if ((x2 & 1) == 1) {
            return x2 == 1 ? (d / 2) - 1 : Frame - d;
            }
        d = d * 2;
        x2 = x2 / 2;
        }
    return 0;
    }
```

16. Appendix D: Outstanding Issues

The following issues need to be addressed.

Issue	Description
Signature	No examples are given of signing a container. Need individual, chain, tree. Leave notarized for notary draft.
Indexing	No examples are given of indexing a container
Archive	Should include a file archive example
File Path	Mention the file path security issue in the security considerations
Security Considerations	Write Security considerations
AES-GCM	Switch to using AES GCM in the examples
KMAC	Switch to using KMAC for KDF
Witness	Complete handling of witness values.
Schema	Complete the schema documentation

Table 1

17. Normative References

[draft-hallambaker-jsonbcd]
Hallam-Baker, P., "Binary Encodings for JavaScript Object Notation: JSON-B, JSON-C, JSON-D", Work in Progress, Internet-Draft, draft-hallambaker-jsonbcd-22, 20 April 2022, <https://datatracker.ietf.org/doc/html/draft-hallambaker-jsonbcd-22>.

[draft-hallambaker-mesh-architecture]
Hallam-Baker, P., "Mathematical Mesh 3.0 Part I: Architecture Guide", Work in Progress, Internet-Draft, draft-hallambaker-mesh-architecture-20, 20 April 2022, <https://datatracker.ietf.org/doc/html/draft-hallambaker-mesh-architecture-20>.

[draft-hallambaker-mesh-security]
Hallam-Baker, P., "Mathematical Mesh 3.0 Part IX Security Considerations", Work in Progress, Internet-Draft, draft-hallambaker-mesh-security-09, 20 April 2022, <https://datatracker.ietf.org/doc/html/draft-hallambaker-mesh-security-09>.

[draft-hallambaker-mesh-udf]
Hallam-Baker, P., "Mathematical Mesh 3.0 Part II: Uniform Data Fingerprint.", Work in Progress, Internet-Draft, draft-hallambaker-mesh-udf-16, 20 April 2022, <https://datatracker.ietf.org/doc/html/draft-hallambaker-mesh-udf-16>.

[IANAJOSE] "[Reference Not Found!]".

[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>.

[RFC2315] Kaliski, B., "PKCS #7: Cryptographic Message Syntax Version 1.5", RFC 2315, DOI 10.17487/RFC2315, March 1998, <https://www.rfc-editor.org/rfc/rfc2315>.

[RFC3394] Schaad, J. and R. Housley, "Advanced Encryption Standard (AES) Key Wrap Algorithm", RFC 3394, DOI 10.17487/RFC3394, September 2002, <https://www.rfc-editor.org/rfc/rfc3394>.

[RFC4880] Callas, J., Donnerhacke, L., Finney, H., Shaw, D., and R. Thayer, "OpenPGP Message Format", RFC 4880, DOI 10.17487/RFC4880, November 2007, <https://www.rfc-editor.org/rfc/rfc4880>.

[RFC4949] Shirey, R., "Internet Security Glossary, Version 2", FYI 36, RFC 4949, DOI 10.17487/RFC4949, August 2007, <https://www.rfc-editor.org/rfc/rfc4949>.

[RFC5869] Krawczyk, H. and P. Eronen, "HMAC-based Extract-and-Expand Key Derivation Function (HKDF)", RFC 5869, DOI 10.17487/RFC5869, May 2010, <https://www.rfc-editor.org/rfc/rfc5869>.

[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>.

[RFC7159] Bray, T., "The JavaScript Object Notation (JSON) Data Interchange Format", RFC 7159, DOI 10.17487/RFC7159, March 2014, <https://www.rfc-editor.org/rfc/rfc7159>.

[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/rfc/rfc7515>.

[RFC7516] Jones, M. and J. Hildebrand, "JSON Web Encryption (JWE)", RFC 7516, DOI 10.17487/RFC7516, May 2015, <https://www.rfc-editor.org/rfc/rfc7516>.

[RFC7517] Jones, M., "JSON Web Key (JWK)", RFC 7517, DOI 10.17487/RFC7517, May 2015, <https://www.rfc-editor.org/rfc/rfc7517>.

[RFC7518] Jones, M., "JSON Web Algorithms (JWA)", RFC 7518, DOI 10.17487/RFC7518, May 2015, <https://www.rfc-editor.org/rfc/rfc7518>.

18. Informative References

[BLOCKCHAIN] Chain.com, "Blockchain Specification".

[Davis2001] Davis, D., "Defective Sign & Encrypt in S/MIME, PKCS#7, MOSS, PEM, PGP, and XML", May 2001.

[RFC5652] Housley, R., "Cryptographic Message Syntax (CMS)", STD 70, RFC 5652, DOI 10.17487/RFC5652, September 2009, <https://www.rfc-editor.org/rfc/rfc5652>.

[ZIPFILE] PKWARE Inc, "APPNOTE.TXT - .ZIP File Format Specification", October 2014.