Data At Rest Encryption Part 2: DARE Container

DARE Data Container

<series>draft-hallambaker-dare-container

<stream>ietf

<status>informational

<ipr>trust200902

<layout tor="true">

<author>Phillip Hallam-Baker

<surname>Hallam-Baker

<initials>P. M.

<firstname>Phillip

<organization>Comodo Group Inc.

<email>[philliph@comodo.com](mailto:philliph@comodo.com)

<also>http://mathmesh.com/Documents/draft-hallambaker-dare-container.html

This document describes DARE Container, a message and file syntax that allows an append-only sequence of data frames to be represented with cryptographic integrity, signature and encryption enhancements. The format supports data integrity checks using digest chains and Merkle trees. The simplest supports efficient write operations and efficient read operations in either the forward or reverse direction. Support for efficient random-access reads may be provided through the use of binary trees or index records appended to the end of the file.

# Introduction

DARE Container 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. DARE Container was developed in response to needs that arose out of the design of the Mathematical Mesh <info="draft-hallambaker-mesh-architecture"/>. It is built on the binary encodings of JSON data objects, JSON-B and JSON-C <norm="draft-hallambaker-jsonbcd"/> and the DARE Message format <norm="draft-hallambaker-dare-message"/>.

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.

## Container Format

A DARE Container 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:

<figuresvg="../Images/jbcdContainer.svg">JBCD Bidirectional Frame

Each frame contains a single DARE Message 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 are 64 bit clean allowing for containers of up to 18 exabytes to be written.

Five container types are currently specified but this may be reduced if the Digest and Tree types are withdrawn. These are:

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.

## Write

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

## Read Access

The use of reverse length indicators allows DARE containers to support efficient sequential access in either the forward or reverse directions.

Random access to any part of a file MAY be supported by means of a binary tree index and/or an index record providing direct access to any part of the file.

## Encryption and Authentication

Frame payloads and associated attributes MAY be encrypted and/or authenticated using the approach described in <norm="draft-hallambaker-dare-message"/>.

*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.

The only restriction on the use of incremental encryption is that the frame containing the key exchange information MUST precede the frames that reference the exchange parameters.

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.

Each DARE Message and by extension, each DARE Container 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.

## 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 Message, 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.

## 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.

It is also possible but not necessarily advisable to perform such a redaction in-place provided that the redaction process does not increase the length of any individual frame and that appropriate provision is made for file locking to prevent conflicts and to provide for safe resumption should an interruption occur during the process.

## 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 Container is designed to provide a superset of the capabilities of existing message and file syntaxes, including:

* Cryptographic Message Syntax <info="RFC5652"/> defines a syntax used to digitally sign, digest, authenticate, or encrypt arbitrary message content.
* The.ZIP File Format specification <info="ZIPFILE"/> developed by Phil Katz.
* The BitCoin Block chain <info="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.

## 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 Container 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 Container 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.

# Definitions

## Related Specifications

DARE Container makes use of the following related standards and specifications.

Encoding

Content frame headers are encoded using JavaScript Object Notation (JSON) <norm="RFC7159"/>, JSON-B or JSON-C <norm="draft-hallambaker-jsonbcd"/>.

Cryptography

The encryption and signature schemes used are based on JSON Web Signature <norm="RFC7515"/> and JSON Web Encryption <norm="RFC7516"/> as applied in DARE Message <norm="draft-hallambaker-dare-message"/>.

## 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 RFC 2119 <norm="RFC2119"/>.

# Container 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(log2(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 Containers 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(n2) 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(log2(n)) search with incremental compilation with approximately double the overhead of the monolithic case.

## Tree

Binary search is supported by means of the TreePosition parameter specified in the FrameHeader. This parameter specifies the value of the immediately preceding apex.

Calculation of the immediately preceding apex is most easily described by representing the array index in binary with base of 1 (rather than 0). An array index that is a power of 2 (2, 4, 8, 16, etc.) will be the apex of a complete tree. Every other array index has the value of the sum of a set of powers of 2 and the immediately preceding apex will be the value of the next smallest power of 2 in the sum.

For example, to find the immediately preceding apex for frame 5, we add 1 to get 6. 6 = 4 + 2, so we ignore the 2 and the preceding frame is 4.

The values of Tree Position are shown for the first 8 frames in figure xx below:

<figuresvg="../Images/MerkleTree1.svg">Merkle Tree Integrity check

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

## Position Index

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

## 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.

# 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.

## 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:

<figuresvg="../Images/hashchain.svg">Hash chain integrity check

The value of the chain digest for the first frame (frame 0) is *H(H(null)+H(Payload0))*, 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(Payloadn-1 +H(Payloadn))*, where *A+B* stands for concatenation of the byte sequences *A* and *B*.

## 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(log2n)* 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.

<include=..\Generated\ContainerEntry.md>

## Signature

Payload data MAY be signed using a JWS <norm="RFC7515"/> as applied in the DARE Message format <norm="draft-hallambaker-dare-message"/>.

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.

# Security Considerations

# IANA Considerations

# Acknowledgements

# Appendix A: Examples and Test Vectors

<include="..\Examples\ExamplesContainer.md">

# Appendix B

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;   
 }