Internet Engineering Task Force
Internet-Draft
Intended status: Experimental
Expires: June 2, 2019

Gwerder

November 29, 2018

MessageVortex Protocol
draft-gwerder-messagevortexmain-00

Abstract

MessageVortex Protocol specifies messages embedded within existing transfer protocols such as SMTP or XMPP to send them anonymously from peer to peer.

The protocol outperforms other protocols by decoupling transport from the final transmitter and receiver party. There is no trust put into any infrastructure except for the infrastructure of the sending and receiving party of a message. The Message flow is entirely selected by the creator of the routing block. Routing nodes gain no non-obvious knowledge about messages even when collaborating. Third-party anonymity is always achieved. Furthermore, one out of sender and receiver anonymity may be achieved.

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 June 2, 2019.

Copyright Notice

Copyright (c) 2018 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents

(https://trustee.ietf.org/license-info) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction 4
1.1. Requirements Language 5
1.2. Protocol Specification 5
1.3. Number Specification 5
2. Entities Overview 5
2.1. Node 5
2.1.1. NodeSpec 5
2.1.1.1. NodeSpec for SMTP nodes 6
2.1.1.2. NodeSpec for XMPP nodes 6
2.2. Peer Partner 6
2.3. Encryption keys 6
2.3.1. Identity Keys 7
2.3.2. Peer Key 7
2.3.3. Sender Key 7
2.4. Vortex Message 7
2.5. Message 8
2.6. Key and MAC specifications and usages 8
2.6.1. Asymmetric Keys 9
2.6.2. Symmetric Keys 9
2.7. Blocks 9
2.8. Transport Address 9
2.9. Identity 10
2.9.1. Peer Identity 10
2.9.2. Ephemeral Identity 10
2.9.3. Official Identity 10
2.10. Workspace 10
3. Layer Overview 11
3.1. Transport Layer 11
3.2. Blending Layer 11
3.3. Routing Layer 12
3.4. Accounting Layer 12
4. Vortex Message 12
4.1. Overview 12
4.2. Message Prefix Block (MPREFIX) 13
4.3. Inner Message Block 13
4.3.1. Control Prefix Block 13
4.3.2. Control Blocks 14
4.3.2.1. Header Block 14

4.3.2.2. Routing Block 14
4.3.3. Payload Block 15
5. General notes 15
5.1. Supported Symmetric Ciphers 15
5.2. Supported Asymmetric Ciphers 15
5.3. Supported MACs 16
5.4. Supported Paddings 16
5.5. Supported Modes 16
6. Blending 17
6.1. Blending in Attachments 17
6.1.1. PLAIN embedding into attachments 17
6.1.2. F5 embedding into attachments 17
6.2. Blending into an SMTP layer 18
6.3. Blending into an XMPP layer 18
7. Routing 18
7.1. Vortex Message Processing 18
7.1.1. Processing of incoming Vortex Messages 18
7.1.2. Processing of Routing Blocks in Workspace 19
7.1.3. Processing of Outgoing MessageVortex Messages 21
7.2. Header Requests 21
7.2.1. Request New Ephemeral Identity 21
7.2.2. Request Message Quota 21
7.2.3. Request Increase of Message Quota 21
7.2.4. Request Transfer Quota 22
7.2.5. Query Quota 22
7.2.6. Request Capabilities 22
7.2.7. Request Nodes 22
7.2.8. Request Identity Replace 23
7.3. Special Blocks 23
7.3.1. Error Block 23
7.3.2. Requirement Block 23
7.3.2.1. Puzzle Requirement 24
7.3.2.2. Payment Requirement 24
7.4. Routing Operations 24
7.4.1. Mapping Operation 25
7.4.2. Split and Merge Operations 25
7.4.3. Encrypt and Decrypt Operations 25
7.4.4. Add and Remove Redundancy Operations 25
7.4.4.1. Padding Operation 26
7.4.4.2. Apply Matrix 26
7.4.4.3. Encrypt Target Block 27
7.5. Processing of Vortex Messages 27
8. Accounting 27
8.1. Accounting Operations 27
8.1.1. Time-Based Garbage Collection 27
8.1.2. Time-Based Routing Initiation 28
8.1.3. Routing Based Quota Updates 28
8.1.4. Routing Based Authorization 28

8.1.5. Ephemeral Identity Creation 28
9. Acknowledgments 28
10. IANA Considerations 28
11. Security Considerations 29
12. References 31
12.1. Normative References 31
12.2. Informative References 33
Appendix A. The ASN.1 schema for Vortex messages 34
A.1. The main VortexMessageBlocks 34
A.2. The VortexMessage Operation Structures 37
A.3. The VortexMessage Ciphers Structures 39
A.4. The VortexMessage Request Structures 42
A.5. The VortexMessage Replies Structures 43
A.6. The VortexMessage Requirements Structures 45
A.7. The VortexMessage Helpers Structures 45
A.8. The VortexMessage Additional Structures 46
Author's Address 48

1. Introduction

Anonymization is hard to achieve. Most of the attempts in the past rely on either trust in a dedicated infrastructure or a specialized networking protocol.

Instead of defining a transport layer, MessageVortex piggybacks on other transport protocols. Messages are being transported alongside ordinary messages of that transport protocol. A blending layer picks the messages up, applies local operations to it and resends the new chunks to the next recipients.

A processing node learns as little as possible from the message due to the nature of the operations processed. The onionized structure of the protocol makes it impossible to follow the trace of a message without having control over the processing node itself.

MessageVortex is a protocol which allows sending and receiving messages by using a routing block instead of a destination address. The sender has full control over all parameters of the message flow.

A message is split and reassembled during transmission. Chunks of the message may carry redundant information to avoid service interruptions of the message transit. Decoy traffic and message traffic are not differentiable as the nature of the addRedundancy operation allows each generated part to be a message part or decoy. Any routing node is thus unable to differentiate between the message and decoy traffic.

Any Receiver knows after processing whether a message is destined for it (it creates a chunk with ID 1) or other nodes (processing instructions only). Due to the missing keys, no other node may do this processing.

This RFC starts with the general terminology (see Section 2). Next, a general overview of the process is given (see Section 3). Lastly, the subsequent sections describe the details of the protocol.

1.1. 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].

1.2. Protocol Specification

Appendix A specifies all relevant parts of the protocol in ASN.1 (see [CCITT.X680.2002] and [CCITT.X208.1988]). The blocks are, if not otherwise mentioned, DER encoded.

1.3. Number Specification

All numbers within this document are, if not suffixed, decimal numbers. Numbers suffixed with a small letter 'h' followed by two hex digits are octets in hex writing. A blank in ASCII (' ') is written as 20h and a capital 'K' in ASCII as 4Bh.

2. Entities Overview

Within this document, we refer to the entities as defined below.

2.1. Node

We use the term 'node' to describe any system connected to other nodes, and supporting the MessageVortex Protocol. A 'node address' is typically an email address, an XMPP address, or any other transport protocol identity supporting the MessageVortex protocol. Any address SHOULD include a public part of an 'identity key' allowing to transmit messages safely. One or more addresses MAY belong to the same node.

2.1.1. NodeSpec

Every addressable node is addressed in a unified format stored in a nodeSpec block. This RFC specifies transport layers for XMPP and SMTP. Adding transport layers requires to write an extension to this RFC.

2.1.1.1. NodeSpec for SMTP nodes

Typically a node specification is specified with the ASN.1 block NodeSpec (see Appendix A.7). To allow addressing of a vortex node with an ordinary client an alternative representation SHOULD be supported as defined below as smtpAlternateSpec (specification noted in ABNF as specified in [RFC5234]).

```
localPart         = <local part of address>
domain            = <domain part of address>
email             = localPart "@" domain
keySpec           = <BASE64 encoded AsymmetricKey; DER representation>;
smtpAlternateSpec = localPart ":" keySpec "@" domain
```

2.1.1.2. NodeSpec for XMPP nodes

Typically a node specification is specified with the ASN.1 block NodeSpec. To allow addressing of a vortex node with an ordinary client, an alternative representation SHOULD be supported as defined below as jidAlternateSpec (specification noted in ABNF as specified in [RFC5234]).

```
localPart         = <local part of address>
domain            = <domain part of address>
jid               = localPart "@" domain [ "/" resourcepart ]
keySpec           = <BASE64 encoded AsymmetricKey; DER representation>;
jidAlternateSpec = localPart ":" keySpec "@" domain [ "/" resourcepart
```

Please note that the ":" character (3Ah) is explicitly excluded in [RFC7622] in the localpart.

2.2. Peer Partner

We use the term 'peer partner' as two or more message sending or receiving entities. One of these peer partners sends a message, and all other peer partners receive one or more messages. Peer partners are message specific. Every peer partner always connects directly to a node.

2.3. Encryption keys

There are several keys required for a Vortex message. For identities and ephemeral identities (see below) we use asymmetric keys. For message encryption, we use symmetric keys.

2.3.1. Identity Keys

Every participant of the network has an asymmetric key. These keys SHOULD be either EC keys with a minimum length of 384 bits or RSA keys with a minimum length of 2048 bits.

The public key needs to be known by all parties writing to or through that node.

2.3.2. Peer Key

Peer keys are symmetrical keys transmitted with a Vortex message. Peer keys are always known to the node sending a message, the node receiving the message from the sender, and to the creator of the routing block.

A peer key is included in the identity block of a Vortex message and in the building instructions for a Vortex message (see RoutingCombo in Appendix A).

2.3.3. Sender Key

The sender key is a symmetrical key protecting the identity and routing block of a Vortex message. It is encrypted with the receiving peer key and prefixed to the identity block. This key decouples further identity and processing information from the previous key.

A sender key is known to precisely one peer of a Vortex message and the creator of the routing block.

2.4. Vortex Message

We use the term 'Vortex message' for a single transmission between two routing layers. A 'blended Vortex message' describes a Vortex message which has been adapted to the transport layer by the blending layer (see Section 3).

A full vortex message contains the following items:

- The peer key (encrypted with the host key of the node; stored in a PrefixBlock) protecting the inner Vortex message (innerMessageBlock).
- The small padding guarantees that a replayed routing block with different content does not look alike.

- The sender key (encrypted with the host key of the node) protecting the identity and routing block.
- The identity block (protected by the sender key) containing information about the ephemeral identity of the sender, replay protection information, header requests (optional) and a requirement reply (optional).
- The routing block (protected by the sender key) containing information on how subsequent messages are processed, assembled and blended.
- The payload block (protected by the peer key) contains payload chunks for processing.

2.5. Message

A Message is a content to be transmitted from one sender to the recipient. The Sender uses a routing block to achieve this which is either built by him or provided by the receiver. A Message may be anonymous. There are however different degrees of anonymity:

- If the sender of a message is not known to anyone else except the sender, we refer to that as 'sender anonymity.'
- If the receiver of a message is not known to anyone else except the receiver, we refer to that as 'receiver anonymity.'
- If an attacker is unable to determine content, original sender, and final receiver, we refer to that as third-party anonymity.
- If a sender or a receiver may be determined as "one of a set of <k> entities we refer to it as k-anonymity (for more about this see [KAnon]).

A message is always MIME encoded as specified in [RFC2045].

2.6. Key and MAC specifications and usages

MessageVortex uses a unique encoding for keys. It is designed to be small and flexible while maintaining a specific base structure.

The following key structures exist:

- SymmetricKey
- AsymmetricKey

MAC does not need a complete structure containing specs and value. Instead only a MacAlgorithmSpec is available. The following sections outline the constraints when specifying parameters for these structures. A node MUST NOT specify any parameter more than once.

If a crypto mode is specified requiring an IV, a node MUST provide the IV when specifying the key.

2.6.1. Asymmetric Keys

Nodes use asymmetric keys for identifying peer nodes (identities) and encrypting symmetric keys (for later de-/encryption of payload or blocks). All asymmetric keys MUST contain a key type specifying a strictly normed key. They MUST contain a public part of the key encoded as X.509 container and a private key as specified in PKCS#8 wherever possible.

RSA and EC keys must contain a keysize parameter. All asymmetric keys SHOULD contain a padding parameter. A node SHOULD assume PKCS#1 if no padding is specified.

NTRU specification MUST provide parameters "n", "p", and "q". McEliece specification MUST contain parameters "n", "k", and "t".

2.6.2. Symmetric Keys

Nodes use symmetric keys for encrypting payload and control blocks. All symmetric keys MUST contain a key type specifying a strictly normed key. They MUST contain a key encoded.

A node MUST provide a keysize parameter if the key (or equivalently block) size is not standardized or encoded in the name. All symmetric key specification MUST contain a mode and padding parameter. A node MAY list multiple padding or mode parameters in a ReplyCapability block to give the recipient a free choice.

2.7. Blocks

We use the term ’block’ for an ASN.1 sequence in a transmitted message. We embed messages in the transport protocol. It may have any size.

2.8. Transport Address

We use the term ’transport address’ for the token required to address the next immediate node on the transport layer. An email transport layer would have SMTP addresses such as ’vortex@example.com’ as transport address.

2.9. Identity

2.9.1. Peer Identity

The peer identity may contain the following information of a peer partner:

- A transport address (always) and the public key of this identity (given there is no recipient anonymity)
- The routing blocks for contacting the identity (optional; this block is only required in the case of recipient anonymity)
- The private key (only known by the owner of the identity)

2.9.2. Ephemeral Identity

Ephemeral identities are temporary identities created on a single node. These identities MUST NOT relate to any other identity. They allow bookkeeping for a node. To each ephemeral identity is a workspace assigned. Every ephemeral identity may have multiple quotas assigned.

2.9.3. Official Identity

An official identity may have the following items assigned:

- Routing blocks to be used to reply to the node.
- A list of assigned ephemeral identities on all other nodes and their projected quotas.
- A list of known nodes and the respective node identity

2.10. Workspace

Every official or ephemeral identity has a workspace. A workspace consists of the following elements:

- Zero or more routing blocks to be processed
- Slots for payload block sequentially numbered. Every slot...
  - MUST contain a numerical ID identifying the slot
  - MAY contain a payload content

* If a block contains a payload, it MUST contain a validity period.

3. Layer Overview

The protocol is designed in four layers as shown in Figure 1

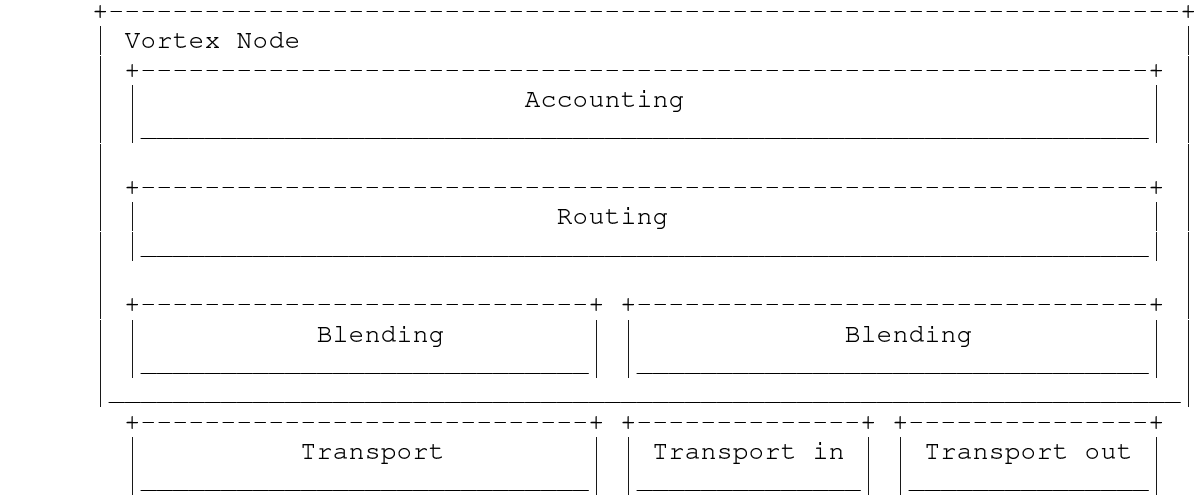

Figure 1: Layer overview

Every participating node MUST implement the layers blending, routing, and accounting. There MUST be at least one incoming and one outgoing transport layer available to a node. All blending layers SHOULD connect to respective Transport layers for sending and receiving packets.

3.1. Transport Layer

The transport layer embeds the blended MessageVortex packets into the data stream of the existing transport layer protocol.

The transport layer infrastructure SHOULD NOT be specific to anonymous communication and should contain significant parts of non-MessageVortex traffic.

3.2. Blending Layer

The blending layer embeds MessageVortex packets into the transport layer data stream and extracts MessageVortex packets from the transport layer.

3.3. Routing Layer

The Routing Layer expands information contained in MessageVortex packets, processes them, and passes generated packets to the respective Blending Layer.

3.4. Accounting Layer

The accounting layer keeps track of all ephemeral identities authorized to use a MessageVortex node. Especially it verifies the available quotas to an ephemeral identity.

4. Vortex Message

4.1. Overview

Figure 2 shows a Vortex message. The enclosed sections denote encrypted blocks. The three to four letter abbreviations denote the key required for decryption. The abbreviation k_h stands for the asymmetric host key. sk_p is the symmetric peer key. The receiving node obtains this key by decrypting MPREFIX with its host key k_h. sk_s is the symmetric sender key. When decrypting the MPREFIX block, the node obtains this key. The sender key protects the header and the routing blocks. This key guarantees that the node assembling the message does not know about upcoming identities, operations, and requests. The peer key protects the message including structure from any third party observer.

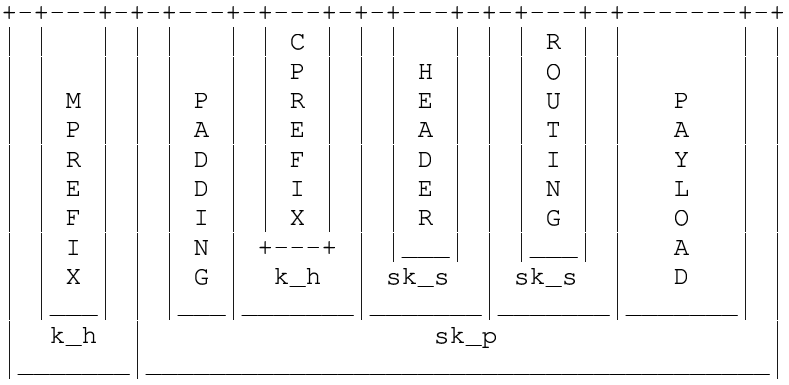

Figure 2: Vortex message overview

4.2. Message Prefix Block (MPREFIX)

The PrefixBlock contains a symmetrical key as defined in Appendix A.1 and is encrypted using the host key of the receiving peer host. The symmetric key used MUST be one out of the set advertised by a CapabilitiesReplyBlock (see Section 7.2.6). A node MAY choose any parameters omitted in the CapabilitiesReplyBlock freely (unless stated otherwise in Section 7.2.6). A node SHOULD avoid sending unencrypted PrefixBlocks. A prefix block MUST contain the same forward-secret as the other prefix, the routing block, and the header block. A host MAY reply to a message with an unencrypted message block. Any reply to a message SHOULD be encrypted.

The sender MUST choose a key which may be encrypted with the host key using the padding advertised by the CapabilitiesReplyBlock.

4.3. Inner Message Block

A node MUST always encrypt (with the symmetric key of the PrefixBlock) an InnerMessageBlock. The encryption hides the inner structure of the message. The InnerMessageBlock SHOULD always accommodate four or more payload chunks.

An InnerMessageBlock always starts with a padding block. This padding guarantees that when using the same routing block multiple times, its binary structure is not repeated throughout the messages with the same routing block. The padding MUST be the first 16 bytes of the first four non-empty payload chunks (PayloadChunks). If a payload chunk is shorter than 16 bytes, the content of the padding SHOULD be filled with zero-valued bytes (00h) at very end up to the required number of bytes. An inner message block (InnerMessageBlock) SHOULD contain at least four payload chunks sized 16 bytes or bigger. If there are less than four payload chunks, then the padding MUST contain a random sequence of 16 bytes for the missing payload chunks. A node MUST NOT reuse random sequences.

An InnerMessageBlock contains so-called forwardSecrets. This random number MUST be the same in the HeaderBlock, the RoutingBlock, and the PrefixBlock. Nodes receiving Messages containing non-matching forwardSecrets MUST discard these messages, and SHOULD NOT send an error message.

4.3.1. Control Prefix Block

Control prefix block (CPREFIX) and MPREFIX block share the same structure and logic. It contains the sender key sk_s. If an MPREFIX block was unencrypted, a node MAY omit the CPREFIX block. An omitted

CPREFIX block results in unencrypted control blocks (HeaderBlock and RoutingBlock).

A prefix block MUST contain the same forwardSecret as the other prefix, the routing block, and the header block.

4.3.2. Control Blocks

The control blocks contain the core information to process the payload. It contains a HeaderBlock and a RoutingBlock.

4.3.2.1. Header Block

The header block (see HeaderBlock in Appendix A) contains the following information:

- It MUST contain the local ephemeral identity of the routing block builder.
- It MAY contain header requests.
- It MAY contain the solution to a PuzzleRequired block previously opposed in a header request.

The list of header requests MAY one of the following:

- be empty
- contain a single identity create request (HeaderRequestIdentity)
- contain a single increase quota request

If a header block violates these rules, then a node MUST NOT reply to any header requests. Payload and routing blocks SHOULD still be added to the workspace and processed, given the message quota is not exceeded.

4.3.2.2. Routing Block

The routing block (see RoutingBlock in Appendix A) contains the following information:

- It MUST contain a serial number uniquely identifying the routing block of this user. A serial number MUST be unique during the lifetime of a routing block.
- It MUST contain the same forward secret as the two prefix blocks and the header block.

- It MAY contain assembly and processing instructions for subsequent messages.
- It MAY contain a reply block for messages assigned to the owner of the identity.

4.3.3. Payload Block

Each InnerMessageBlock containing routing information SHOULD contain at least four PayloadChunks.

5. General notes

The MessageVortex protocol is a modular protocol. It allows using different encryption algorithms. For operation, a Vortex node SHOULD always support at least two completely different (i.e. relying on two different mathematical problems) types of algorithms, paddings, or modes.

5.1. Supported Symmetric Ciphers

A node MUST support the following symmetric ciphers:

- AES128 (see [FIPS-AES] for AES implementation details)
- AES256
- CAMELLIA128 (see [RFC3657] chapter 3 for Camellia implementation details)
- CAMELLIA256

A node SHOULD support any standardized, bigger key size than the smallest key.

A node MAY support Twofish ciphers (see [TWOFISH]).

5.2. Supported Asymmetric Ciphers

A node MUST support the following asymmetric ciphers:

- RSA (key sizes bigger or equal to 2048) ([RFC8017])
- ECC (named curves secp384r1, sect409k1, secp521r1) (see [SEC1])

5.3. Supported MACs

A node MUST support the following Message Authentication Codes (MAC):

- SHA256 (see [ISO-10118-3] for SHA implementation details)
- RipeMD160 (see [ISO-10118-3] for RIPEMD implementation details)

A node SHOULD support the following MACs:

- SHA512
- RipeMD256
- RipeMD512

5.4. Supported Paddings

A node MUST support the following paddings specified in [RFC8017]:

- PKCS1 (see [RFC8017])
- PKCS7 (see [RFC5958])

5.5. Supported Modes

A node MUST support the following modes:

- CBC (see [RFC1423]). The used IV must be of equal length as the key)
- EAX (see [EAX])
- GCM (see [RFC5288])
- NONE (only used in special cases. See Section 11)

A node SHOULD NOT use the following modes:

- NONE (Except as stated when using the addRedundancy function
- ECB

A node SHOULD support the following modes:

- CTR ([RFC3686])
- CCM ([RFC3610])

- OCB ([RFC7253])
- OFB ([MODES])

6. Blending

Each node supports a fixed set of blending capabilities. They may be different for incoming and outgoing messages.

The following sections describe the blending mechanism. There are currently two blending layers specified. One layer specification for the simple mail transfer protocol (SMTP; See [RFC5321]) and one for the Extensible Messaging and Presence Protocol (XMPP; See [RFC6120]).

6.1. Blending in Attachments

There are two types of blending supported when using attachments.

- Plain binary encoding with offset (PLAIN)
- Embedding with F5 in an image (F5)

A node MUST support PLAIN blending for reasons of interoperability. A node MAY support blending using F5.

6.1.1. PLAIN embedding into attachments

A blending layer embeds a VortexMessage in a carrier file with an offset for PLAIN blending. For replacing a file start, a node MUST use the offset 0. The routing node MUST choose the payload file for the message. A routing node SHOULD use a credible payload type (e.g., MIME type) with high entropy. It furthermore SHOULD prefix a valid header structure to avoid easy detection of the Vortex message. A routing node SHOULD use a valid footer, if any, to a payload file to improve blending.

A node SHOULD offer at least one PLAIN blending method for incoming Vortex messages. A node may offer multiple offsets for incoming Vortex messages.

6.1.2. F5 embedding into attachments

For F5, a blending layer embeds a VortexMessage into a jpeg file according to [F5]. The password for blending may be publicly known. A routing node MAY advertise multiple passwords. The use of F5 adds roughly a tenfold of transfer volume to the message. A routing block building node SHOULD only use F5 blending where appropriate.

6.2. Blending into an SMTP layer

Email messages containing messages MUST be encoded with Multipurpose Internet Mail Extensions (MIME) as specified in [RFC2045]. All nodes MUST support BASE64 encoding. A node MUST test all sections of a MIME message for the presence of a VortexMessage.

A vortex message is present if a block containing the peer key at the known offset of any MIME part decodes correctly.

A node SHOULD support SMTP blending for sending and receiving. For sending SMTP as specified in [RFC5321] must be used. TLS layers MUST always be applied when obtaining messages using POP3 (as specified in [RFC1939] and [RFC2595]) or IMAP (as specified in [RFC3501]). Any SMTP connection MUST employ a TLS encryption when passing any credentials.

6.3. Blending into an XMPP layer

For interoperability, an implementation SHOULD provide XMPP blending.

Blending into XMPP traffic is done using the [XEP-0234] extension of the XMPP protocol.

PLAIN and F5 blending is acceptable for this transport layer.

7. Routing

7.1. Vortex Message Processing

7.1.1. Processing of incoming Vortex Messages

An incoming message is considered unauthenticated at first. A node should consider a VortexMessage as authenticated as soon as the ephemeral identity is known and is not temporary.

For an unauthenticated message the following rules apply:

- A node MUST ignore all Routing blocks.
- A node MUST ignore all Payload blocks.
- A node SHOULD accept identity creation requests in unauthenticated messages.
- A node MUST ignore any other header request except identity creation requests.

- A node MUST ignore all identity creation requests which belong to an already existing identity.

A message is considered authenticated as soon as the identity used in the header block is known and not temporary. A node MUST NOT treat a message as authenticated if the specified maximum number of replays have been reached. For authenticated messages the following rules apply:

- A node MUST ignore identity creation requests.
- A node MUST replace the current reply block with the reply block provided in the routing block (if any). The node MUST keep the reply block if no reply block is provided.
- A node SHOULD process all header requests.
- A node SHOULD add all routing blocks to the workspace.
- A node SHOULD add all payload blocks to the workspace.

A routing node MUST decrement the message quota by one if a received message is authenticated and contains at least one payload block. If a message is identified as duplicate acording to the reply protection, a node MUST NOT decrement the message quota.

7.1.2. Processing of Routing Blocks in Workspace

A routing workspace consists of the following items:

- The identity linked to it (This determines the lifetime of the workspace).
- The routing combos (RoutingCombo) linked to it.
- A payload chunk space. Multiple subspaces are available within this space:
  - ID 0 represents a message to be embedded (when reading) or a message to be extracted to the user (when written).
  - ID 1 to ID maxPayloadBlocks represents the payload chunk slots in the target message.
  - All blocks between ID maxPayloadBlocks+1 to ID 32767 belong to a temporary routing block specific space.

* All blocks between ID 32768 to ID 65535 belong to a shared space available to all operations of this identity.

The accounting layer typically triggers processing. It represents either a cleanup action or a routing event. A cleanup event deletes the following information from all workspaces:

- All processed routing combos.
- All routing combos with expired usagePeriod.
- All payload chunks when exceeded their maxProcess time.
- All expired objects.
- All expired puzzles.
- All expired identities.
- All expired replay protections.

Note that maxProcessTime reflects the number of seconds since the arrival of the last octet of the message at the transport layer facility. A node SHOULD NOT take additional processing time (e.g., for anti-UBE or anti-virus) into account.

The accounting layer triggers routing events. The trigger occurs at least minProcessTime after the last octet of the message arrived at the routing layer. A node SHOULD choose the latest possible moment in such a way that the peer node receives the last octet of the assembled message before maxProcessTime is reached. The calculation of the last point in time where a message may be set SHOULD always assume that the target node is working. A sending node SHOULD choose the time within these bounds randomly. An accounting layer MAY trigger multiple routing combos in bulk to further obfuscate identity of a single transport message.

First, the processing node escapes the payload chunk at ID 0 if needed (non-special block starting with a backslash). Next, it executes all processing instructions of a routing combo in the sequence specified. If an instruction fails, the block at the target ID of the operation remains unchanged. The routing layer proceeds with the subsequent processing instructions, ignoring the error. For a detailed description of the operations see Section 7.4. If a node succeeds in building at least one payload chunk, a VortexMessage is composed and passed to the blending layer.

7.1.3. Processing of Outgoing MessageVortex Messages

The blending layer MUST then compose a transport layer message according to the specification provided in the routing combo. It SHOULD choose any decoy message or steganographic carrier in such a way that the dead parrot syndrome as specified in [DeadParrot] is avoided.

7.2. Header Requests

Header requests are control requests for the anonymization system. Messages with requests or replies only MUST NOT affect any quota.

7.2.1. Request New Ephemeral Identity

Requesting a new ephemeral identity is done by sending a message containing a header block with the new identity and an identity creation request (HeaderRequestIdentity) to a node. The node MAY send an error block (see Section 7.3.1) if rejecting the request.

If a node accepts an identity creation request, it MUST send a reply. To accept a request without a requirement, an accepting node MUST send back a special block containing "no error". To accept a block with a requirement, an accepting node MUST send a special block containing a requirement block.

7.2.2. Request Message Quota

Any valid ephemeral identity may request to raise the current message quota to a specific value at any time. The request MUST include a reply block in the header. The request may contain other parts. If a requested value is lower than the current quota, the node SHOULD NOT refuse the quota request and SHOULD send a "No Error" status.

A node SHOULD reply to a HeaderRequestIncreaseMessageQuota request (see Appendix A) of a valid ephemeral identity. The reply MUST include a requirement, an error message or a "No Error" status message.

7.2.3. Request Increase of Message Quota

A node may request to increase the current message quota. The increase is requested by sending a HeaderRequestIncreaseMessageQuota request to the routing node. The value specified within the node is the new quota. HeaderRequestIncreaseMessageQuota requests MUST include a reply block. A node SHOULD NOT use a previously sent MURB to reply.

If the requested quota is higher than the current quota, then the node SHOULD send a "no error" reply. If the requested quota is not accepted, the node SHOULD send a requestedQuotaOutOfBand reply.

A node accepting the request MUST send a RequirementBlock or a "no error block".

7.2.4. Request Transfer Quota

Any valid ephemeral identity may request to raise the current transfer quota to a specific value at any time. The request MUST include a reply block in the header. The request may contain other parts. If a requested value is lower than the current quota, the node SHOULD NOT refuse the quota request and SHOULD send a "No Error" status.

A node SHOULD reply to a HeaderRequestIncreaseTransferQuota request (see Appendix A) of a valid ephemeral identity. The reply MUST include a requirement, an error message, or a "No Error" status message.

7.2.5. Query Quota

Any valid ephemeral identity may request the current message and transfer quota. The request MUST include a reply block in the header. The request may contain other parts.

A node MUST reply to a HeaderRequestQueryQuota request (see Appendix A). The reply MUST include the current message quota and the current message transfer quota. The reply to this request MUST NOT include a requirement.

7.2.6. Request Capabilities

Any node MAY request the capabilities of another node. The capabilities include all information necessary to create a parseable VortexMessage. Any node SHOULD reply to any encrypted HeaderRequestCapability.

7.2.7. Request Nodes

A node may ask another node for a list of routing node addresses and keys. This request may be used to bootstrap a new node and to add routing nodes increasing the anonymization of a node. The receiving node of such a request SHOULD reply with a requirement (e.g., RequirementPuzzleRequired).

A node SHOULD reply to a HeaderRequest request (see Appendix A) of a valid ephemeral identity. The reply MUST include a requirement, an error message or a "No Error" status message.

7.2.8. Request Identity Replace

This request allows a receiving node to replace an identity with the identity provided in the message. This request is required if an adversary managed to deny the usage of a node (e.g., by deleting the corresponding transport account). Any sending node may recover from such an attack by sending a validly authenticated message to another identity providing the new transport and key details.

A node SHOULD reply to a such a request of a valid known identity. The reply MUST include an error message or a "No Error" status message.

7.3. Special Blocks

Special blocks are payload messages. They reflect messages from one node to another and are not visible to the user. A special block starts with the character sequence '\special' (or 5Ch 73h 70h 65h 63h 69h 61h 6Ch) followed by a DER encoded special block (SpecialBlock). Any non-special message decoding to ID 0 in a workspace starting with a backslash (5Ch) MUST escape all backslashes within the payload chunk with an additional backslash.

7.3.1. Error Block

An error block may be sent as a reply where specified as a payload. The error block is embedded in a special block and sent with any provided reply block. Error messages SHOULD contain the serial number of the offending header block and MAY contain a human-readable text providing additional messages about the error.

7.3.2. Requirement Block

If a node is receiving a requirements block, it MUST assume that the request block has been accepted, has not been processed yet, and will be processed if the contained requirement is met. A node MUST process a request as soon as the requirement is fulfilled. A node MUST resend the request as soon as the requirement is met.

A node MAY reject a request, accept a request without a requirement, accept a request upon payment (RequirementPaymentRequired), or accept a request upon solving a proof of work puzzle (RequirementPuzzleRequired).

7.3.2.1. Puzzle Requirement

If a node requests a puzzle, it MUST send a RequirementPuzzleRequired block. The puzzle requirement is solved, if the node receiving the puzzle is replying with a header block containing the puzzle block and the hash of the encoded block starts with the bit sequence mentioned in the puzzle within the period specified in the field 'valid'.

To solve a puzzle posed by a node a Vortex Message needs to be sent to the requesting node. This Vortex Message MUST contain a header block which includes the puzzle block and MUST have a MAC fingerprint starting with the bit sequence as specified in the challenge. A node calculates the MAC from the unencrypted DER encoded HeaderBlock with the algorithm specified by the node. To meet this requirement, a node adds a proofOfWork field to the HeaderBlock.

7.3.2.2. Payment Requirement

If a node requests a payment, it MUST send a RequirementPaymentRequired block. As soon as the requested fee is paid and confirmed, the requesting node MUST send a "No Error" status message. The usage period 'valid' describes the period in which the payment may be carried out. A node MUST accept the payment if carried out within the 'valid' period but confirmed later. A node SHOULD return allunsolicited payments to the sending address.

7.4. Routing Operations

Routing operations are contained in a routing block and processed either on arrival on a message or when a compiling new message. All Operations are reversible. No Operation is available for generating decoy traffic. For decoy traffic, either encryption of an unpadded block may be used, or the addRedundancy operation.

All payload chunk blocks inherit the validity time from the message routing combos (arrival time + max(maxProcessTime)).

When applying an operation to a source block, the resulting target block inherits the expiry of the of the source block. When having multiple different expiry times, the expiry the furthest in the future will be applied to the target block. If the operation fails, the target expiry remains unchanged.

7.4.1. Mapping Operation

A mapping operation is a straightforward operation mainly used in inOperations of a routing block to map the routing block specific blocks to a permanent workspace.

7.4.2. Split and Merge Operations

The split and merge operations allow splitting and recombining message chunks. A node MUST adhere to these constraints:

- The operation must be applied at an absolute (measuring in bytes) or relative (measured as a float value in the range 0>value>100) position.
- All calculations must be done according to IEEE 754 [IEEE754] and in 64 Bit precision.
- If a relative value is a non-integer result, a floor operation (cutting off all non-integer parts) determines the number of bytes.
- If an absolute value is negative, the size reflected applies to the number of bytes counted from the end of the message chunk.
- If an absolute value is bigger than the number of bytes in a block, all bytes are mapped to the respective target block, and the other target block becomes a zero byte sized block.

An operation MUST fail if relative values are equal to, or below zero. An operation MUST fail if a relative value is equal to or above 100. All floating point operations must be carried out according to [IEEE754] and in 64-bit precision.

7.4.3. Encrypt and Decrypt Operations

Encryption and decryption are executed according to the standards mentioned before. An encryption operation encrypts a block symmetrically and places the result in the target block. The parameters MUST contain required parameters such as IV, padding, or cipher modes. An encryption operation without a valid parameter set MUST fail.

7.4.4. Add and Remove Redundancy Operations

The addRedundancy and removeRedundancy operations are the core operations of the protocol. They may be used to split messages and

distribute message content across multiple routing nodes. The operation is split into three steps.

1. Pad the input block to a multiple of the key block size in the resulting output blocks.

2. Apply a Vandermonde matrix with the given sizes.

3. Encrypt each resulting block with a separate key.

The following sections describe the order of the operations in an addRedundancy operation. For a removeRedundancy operation invert the functions and order.

7.4.4.1. Padding Operation

First, the length of all output blocks including redundancy is calculated. This is done by L=roof((<input block size in bytes>+4)/<encryption block size in bytes>)*<block size in bytes>. The block is prepended with a 32-bit length indicator in bytes at the beginning (little-endian). This length indicator i is calculated by i=<input block size in bytes>*randominteger()*L. The rest of the input block up to length L is padded with random data. If GF(16) is applied, numbers are treated as little-endian representations.

For padding removal, first, the padding i at the start is removed as a little-endian integer. Then, the length of the output block is calculated by applying <output block size in bytes>=i mod <input block size in bytes>

This padding guarantees that each resulting block is as big as the subsequent encryption operation and does not require any further padding.

7.4.4.2. Apply Matrix

Next, the input block is organized in a data matrix D of dimensions inrows,incols where incols=(<number of data blocks>-<number of redundancy blocks>) and inrows=L/(<number of data blocks>-<number of redundancy blocks>). The input block data is distributed in this matrix first across, then down.

Next, we multiply the data matrix D by a Vandermonde matrix V. The V matrix has the number of rows equal to the incols calculated, and columns are equal to the <number of data blocks>. The content of the matrix is formed by v(i,j)=pow(i,j), whereas i reflects the row number starting at 0, and j reflects the column number starting at 0. Please note that calculations noted here have to be carried out in

the GF noted in the respective operation to be successful. The operation results in a matrix A.

7.4.4.3. Encrypt Target Block

Each row vector of A is a new data block which is then encrypted with the corresponding encryption key noted in keys of the addRedundancyOperation. If there are not enough keys available, the keys used for encryption are reused from the beginning after the last key has been used. A routing block builder SHOULD provide enough keys so that all target blocks may be encrypted with a unique key. All encryptions SHOULD NOT use padding.

7.5. Processing of Vortex Messages

The accounting layer triggers processing according to information contained in a routing block in the workspace. All operations MUST be executed in the sequence provided in the routing block. Any failing operation must leave the result block unmodified.

All workspace blocks resulting in IDs 1 to maxPayloadBlock are then added to the message and passed to the blending layer with appropriate instructions.

8. Accounting

8.1. Accounting Operations

The accounting layer has two major kinds of operations:

- Time-based operations (cleanup jobs and initiation of routing)
- Routing triggered operations (updating quotas, authorizing operations, and pickup of incoming messages)

Implementations MUST provide sufficient locking mechanisms to guarantee the integrity of accounting information and workspace at any time.

8.1.1. Time-Based Garbage Collection

The accounting layer SHOULD keep a list of expiry times. As soon as an entry (e.g., payload block, or identity) expires, the respective structure should be removed from the workspace. An implementation MAY choose to remove expired items periodically or when encountering them during normal operation.

8.1.2. Time-Based Routing Initiation

The accounting layer MAY keep a list of any time a routing block is activated. For improved privacy, the accounting layer should use a slotted model where, whenever possible, multiple routing blocks are handled in the same period of time, and the requests to the blending layers are mixed between the transactions.

8.1.3. Routing Based Quota Updates

A node MUST update quotas on the respective operations. It MUST decrease message quota before processing routing blocks in the workspace. A node MUST decrease the message quota after the processing of any header requests.

8.1.4. Routing Based Authorization

The transfer quota MUST be checked and decreased by the number of data bytes in the payload chunks after all header requests have been processed. The message quota MUST be decreased by one on each routing block triggering processing in the workspace.

8.1.5. Ephemeral Identity Creation

Any packet may request the creation of an ephemeral identity. A node SHOULD NOT accept such a request without a costly requirement. The request includes a lifetime of the ephemeral identity. The costs for creating the ephemeral identity SHOULD raise if a longer lifetime is requested.

9. Acknowledgments

Thanks go to my family which did support me with patience and countless hours and to Mark Zeman for his feedback challenging my thoughts and peace.

10. IANA Considerations

This memo includes no request to IANA.

Additional encryption algorithms, paddings, modes, blending layers, or puzzles MUST be added by writing an extension to this or a subsequent RFC. For testing purposes, IDs above 1,000,000 should be used.

11. Security Considerations

The MessageVortex protocol may be understood more as a toolset than a fixed product. Depending on the usage of the toolset anonymity and security are affected. For a detailed analysis see [MVAnalysis].

The primary goals for security within this protocol did rely on the following focus areas:

- Confidentiality
- Integrity
- Availability
- Anonymity
  - 3rd party anonymity
  - sender anonymity
  - receiver anonymity

All these factors are affected by the usage of the protocol. The following sections provide a list of factors affecting the primary goals.

The Vortex protocol does not rely on any encryption on the transport layer. Vortex messages are already encrypted. Confidentiality is not affected by the protection mechanisms of the transport layer.

If a transport layer supports encryption, a Vortex node SHOULD use it to improve the privacy of the message.

Anonymity is affected by the inner workings of the blending layer in many ways. A Vortex message cannot be read by anyone except the peer nodes and the routing block builder, but the presence of a vortex node message may be detected. This may be done either by detecting the typical high entropy of an encrypted file, broken structures of a carrier file, a meaningless content of a carrier file, or the contextless communication of the transport layer with its peer partner. A blending layer SHOULD minimize the possibility of easy detection by minimizing these effects.

A blending layer SHOULD use carrier files with high compression or encryption. Carrier files SHOULD NOT have inner structures so that the payload is comparable to valid content. To achieve undetectability by a human reviewer, a routing block builder should

use F5 blending instead of PLAIN blending. This, however, increases the protocol overhead roughly by a tenfold.

The two layers 'routing' and 'accounting' have the deepest insight into a Vortex message's inner working. They know the immediate peer sender and the peer recipients of all payload chunks. As decoy traffic is generated by combining chunks and applying redundancy calculations upon them, a node can never know whether a malfunction (e.g., when doing a recovery calculation) was intended or not. Therefore a node is unable to tell a failed transaction apart from a terminated transaction. It furthermore cannot tell content apart from decoy traffic.

A routing block builder SHOULD follow the following rules in order not to compromise a Vortex message's anonymity:

- All operations applied SHOULD be credibly involved in a message transfer.
- There should always be a sufficient subset of the result of an addRedundancy operation sent to peers to allow recovery of the data built.
- The anonymity set of a message should be sufficiently large to avoid legal prosecution of all jurisdictional entities involved. It has to be large enough to do so even if a certain amount of the anonymity set cooperates with an adversary.
- Encryption and decryption SHOULD follow whenever possible normal usage. Avoid encrypting a block on a node with one key and decrypting it with a different key on the same or an adjacent node.
- Traffic peaks SHOULD be uniformly distributed within the whole anonymity set.
- A routing block SHOULD be used for a limited number of messages. If used as a message block for the node itself it should be used only once. A block builder SHOULD use the HeaderRequestReplaceIdentity block to update reply routing blocks on a regular base. Implementers should always keep in mind that the same routing block is identifiable as such by its structure.

An active adversary cannot use blocks from other routing block builders for his purposes. He may falsify the result by injecting wrong message chunks or by not sending a message. Such message disruptions may be detected by intentionally routing some information to the routing block builders' node. If the Vortex message does not

carry the information expected the node may safely assume that one of the involved nodes is misbehaving. A block building node MAY calculate reputation for involved nodes over time. A block building node MAY build redundancy paths into a routing block to withstand such malicious nodes.

Receiver anonymity is in danger if the handling of message header and content is not done with care. An attacker might send a bugged message (e.g., with a DKIM or DMARC header) to deanonymize a recipient. Great care has to be taken when handling any other than local references when processing, verifying, or rendering a message.

12. References

12.1. Normative References

[CCITT.X208.1988]
International Telephone and Telegraph Consultative Committee, "Specification of Abstract Syntax Notation One (ASN.1)", CCITT Recommendation X.208, 11 1998.

[CCITT.X680.2002]
International Telephone and Telegraph Consultative Committee, "Abstract Syntax Notation One (ASN.1): Specification of basic notation", 11 2002.

[EAX] Bellare, M., Rogaway, P., and D. Wagner, "The EAX mode of operation", 2011.

[F5] Westfeld, A., "F5 - A Steganographic Algorithm - High Capacity Despite Better Steganalysis", 10 2001.

[FIPS-AES]
Federal Information Processing Standard (FIPS), "Specification for the ADVANCED ENCRYPTION STANDARD (AES)", 11 2011.

[IEEE754] IEEE, "754-2008 - IEEE Standard for Floating-Point Arithmetic", 08 2008.

[ISO-10118-3]
International Organization for Standardization, "ISO/IEC 10118-3:2004 -- Information technology -- Security techniques -- Hash-functions -- Part 3: Dedicated hash-functions", 3 2004.

[MODES] National Institute for Standards and Technology (NIST), "Recommendation for Block Cipher Modes of Operation: Methods and Techniques", 12 2001.

[RFC1423] Balenson, D., "Privacy Enhancement for Internet Electronic Mail: Part III: Algorithms, Modes, and Identifiers", RFC 1423, DOI 10.17487/RFC1423, February 1993, <https://www.rfc-editor.org/info/rfc1423>.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[RFC3610] Whiting, D., Housley, R., and N. Ferguson, "Counter with CBC-MAC (CCM)", RFC 3610, DOI 10.17487/RFC3610, September 2003, <https://www.rfc-editor.org/info/rfc3610>.

[RFC3657] Moriai, S. and A. Kato, "Use of the Camellia Encryption Algorithm in Cryptographic Message Syntax (CMS)", RFC 3657, DOI 10.17487/RFC3657, January 2004, <https://www.rfc-editor.org/info/rfc3657>.

[RFC3686] Housley, R., "Using Advanced Encryption Standard (AES) Counter Mode With IPsec Encapsulating Security Payload (ESP)", RFC 3686, DOI 10.17487/RFC3686, January 2004, <https://www.rfc-editor.org/info/rfc3686>.

[RFC5234] Crocker, D., Ed. and P. Overell, "Augmented BNF for Syntax Specifications: ABNF", STD 68, RFC 5234, DOI 10.17487/RFC5234, January 2008, <https://www.rfc-editor.org/info/rfc5234>.

[RFC5288] Salowey, J., Choudhury, A., and D. McGrew, "AES Galois Counter Mode (GCM) Cipher Suites for TLS", RFC 5288, DOI 10.17487/RFC5288, August 2008, <https://www.rfc-editor.org/info/rfc5288>.

[RFC5958] Turner, S., "Asymmetric Key Packages", RFC 5958, DOI 10.17487/RFC5958, August 2010, <https://www.rfc-editor.org/info/rfc5958>.

[RFC7253] Krovetz, T. and P. Rogaway, "The OCB Authenticated-Encryption Algorithm", RFC 7253, DOI 10.17487/RFC7253, May 2014, <https://www.rfc-editor.org/info/rfc7253>.

[RFC8017] Moriarty, K., Ed., Kaliski, B., Jonsson, J., and A. Rusch, "PKCS #1: RSA Cryptography Specifications Version 2.2", RFC 8017, DOI 10.17487/RFC8017, November 2016, <https://www.rfc-editor.org/info/rfc8017>.

[SEC1] Certicom Research, "SEC 1: Elliptic Curve Cryptography", 05 2009.

[TWOFISH] Schneier, B., "The Twofish Encryptions Algorithm: A 128-Bit Block Cipher, 1st Edition", 03 1999.

[XEP-0234] Peter, S. and L. Stout, "XEP-0234: Jingle File Transfer", 08 2017, <https://xmpp.org/extensions/xep-0234.html>.

12.2. Informative References

[DeadParrot] Houmansadr, A., Burbaker, C., and V. Shmatikov, "The Parrot is Dead: Observing Unobservable Network Communications", 2013, <https://people.cs.umass.edu/~amir/papers/parrot.pdf>.

[KAnon] Ahn, L., Bortz, A., and N. Hopper, "k-Anonymous Message Transmission", 2003.

[MVAnalysis] Gwerder, M., "MessageVortex", 2018, <https://messagevortex.net/devel/messageVortex.pdf>.

[RFC1939] Myers, J. and M. Rose, "Post Office Protocol - Version 3", STD 53, RFC 1939, DOI 10.17487/RFC1939, May 1996, <https://www.rfc-editor.org/info/rfc1939>.

[RFC2045] Freed, N. and N. Borenstein, "Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies", RFC 2045, DOI 10.17487/RFC2045, November 1996, <https://www.rfc-editor.org/info/rfc2045>.

[RFC2595] Newman, C., "Using TLS with IMAP, POP3 and ACAP", RFC 2595, DOI 10.17487/RFC2595, June 1999, <https://www.rfc-editor.org/info/rfc2595>.

[RFC3501] Crispin, M., "INTERNET MESSAGE ACCESS PROTOCOL - VERSION 4rev1", RFC 3501, DOI 10.17487/RFC3501, March 2003, <https://www.rfc-editor.org/info/rfc3501>.

[RFC5321] Klensin, J., "Simple Mail Transfer Protocol", RFC 5321, DOI 10.17487/RFC5321, October 2008, <https://www.rfc-editor.org/info/rfc5321>.

[RFC6120] Saint-Andre, P., "Extensible Messaging and Presence Protocol (XMPP): Core", RFC 6120, DOI 10.17487/RFC6120, March 2011, <https://www.rfc-editor.org/info/rfc6120>.

[RFC7622] Saint-Andre, P., "Extensible Messaging and Presence Protocol (XMPP): Address Format", RFC 7622, DOI 10.17487/RFC7622, September 2015, <https://www.rfc-editor.org/info/rfc7622>.

Appendix A. The ASN.1 schema for Vortex messages

The following sections contain the ASN.1 modules specifying the MessageVortex Protocol

A.1. The main VortexMessageBlocks

```
MessageVortex-Schema DEFINITIONS EXPLICIT TAGS ::=
BEGIN
  EXPORTS PrefixBlock, InnerMessageBlock, RoutingBlock,
          maxID;
  IMPORTS SymmetricKey, AsymmetricKey, MacAlgorithmSpec, CipherSpec
                FROM MessageVortex-Ciphers
          HeaderRequest
                FROM MessageVortex-Requests
          PayloadOperation
                FROM MessageVortex-Operations

          UsagePeriod, BlendingSpec
                FROM MessageVortex-Helpers;

  --****************************************************************
  -- Constant definitions
  --****************************************************************

  -- maximum serial number
  maxSerial                 INTEGER ::= 4294967295
  -- maximum number of administrative requests
  maxNumberOfRequests       INTEGER ::= 8
  -- maximum number of seconds which the message might be delayed
  -- in the local queue (starting from startOffset)
  maxDurationOfProcessing   INTEGER ::= 86400
  -- maximum id of an operation
  maxID                     INTEGER ::= 32767
  -- maximum number of routing blocks in a message
```

```
maxRoutingBlocks          INTEGER ::= 127
-- maximum number a block may be replayed
maxNumberOfReplays        INTEGER ::= 127
-- maximum number of payload chunks in a message
maxPayloadBlocks          INTEGER ::= 127
-- maximum number of seconds a proof of non revocation may be old
maxTimeCachedProof        INTEGER ::= 86400
-- The maximum ID of the workspace
maxWorkspaceId            INTEGER ::= 65535
-- The maximum number of assembly instructions per combo
maxAssemblyInstructions   INTEGER ::= 255

--****************************************************************
-- Block Definitions
--****************************************************************
PrefixBlock ::= SEQUENCE {
  forwardsecret   ChainSecret,
  key             SymmetricKey,
      version                INTEGER OPTIONAL
}

HeaderBlock ::= SEQUENCE {
  -- Public key of the identity representing this transmission
  identityKey        AsymmetricKey,
  -- serial identifying this block
  serial             INTEGER (0..maxSerial),
  -- number of times this block may be replayed (Tuple is
  -- identityKey, serial while UsagePeriod of block)
  maxReplays         INTEGER (0..maxNumberOfReplays),
  -- subsequent Blocks are not processed before valid time.
  -- Host may reject too long retention. Recomended validity
  -- support >=1Mt.
  valid              UsagePeriod,
  -- represents the chained secret which has to be found in
  -- subsequent blocks
  -- prevents reassembly attack
  forwardSecret       ChainSecret,
  --  contains the MAC-Algorithm used for signing
  signAlgorithm       MacAlgorithmSpec,
  -- contains administrative requests such as quota requests
  requests            SEQUENCE (SIZE (0..maxNumberOfRequests))
                      OF HeaderRequest ,
  -- Reply Block for the requests
  requestReplyBlock   RoutingCombo,
  -- padding and identitifier required to solve the cryptopuzzle
  identifier  [12201] PuzzleIdentifier OPTIONAL,
  -- This is for solving crypto puzzles
  proofOfWork [12202] OCTET STRING OPTIONAL
```

```
}

InnerMessageBlock ::= SEQUENCE {
  padding   OCTET STRING,
  prefix    CHOICE {
      plain           [11011] PrefixBlock,
      -- contains prefix encrypted with receivers public key
      encrypted       [11012] OCTET STRING
  },
  identity  CHOICE  {
    -- debug/internal use only
    plain     [11021] HeaderBlock,
    -- contains encrypted identity block
    encyrpted [11022] OCTET STRING
  },
  -- contains signature of Identity [as stored in
  -- HeaderBlock; signed unencrypted HeaderBlock without Tag]
  identitySignature OCTET STRING,
  -- contains routing information (next hop) for the payloads
  routing     CHOICE {
    plain     [11031] RoutingBlock,
    -- contains encrypted routing block
    encyrpted [11032] OCTET STRING
  },
  -- contains the actual payload
  payload     SEQUENCE (SIZE (0..maxPayloadBlocks))
              OF OCTET STRING
}

RoutingBlock ::= SEQUENCE {
  -- contains the prefix to be used
  routing    CHOICE {
    -- debug/internal use only
    plain             [331] SEQUENCE (SIZE (0..maxRoutingBlocks))
                            OF RoutingCombo,
    encrypted         [332] SEQUENCE (SIZE (0..maxRoutingBlocks))
                            OF OCTET STRING
  },
  -- contains the secret of the header block
  forwardSecret     ChainSecret,
  -- contains a routing block which may be used when sending
  -- error messages back to the quota owner
  -- this routing block may be cached for future use
  replyBlock [131]   SEQUENCE {
       murb           RoutingCombo,
       maxReplay      INTEGER,
       validity       UsagePeriod
  } OPTIONAL
```

```
    }

    RoutingCombo ::= SEQUENCE {
      -- contains the period when the payload should be processed
      -- Router might refuse to long queue retention
      -- Recommended support for retention >=1h
      minProcessTime INTEGER (0..maxDurationOfProcessing),
      maxProcessTime INTEGER (0..maxDurationOfProcessing),
      -- The message key to encrypt the message
      messageKey    [401] SymmetricKey OPTIONAL,
      -- contains the next recipient
      recipient     [402] BlendingSpec OPTIONAL,
      -- PrefixBlock encrypted with message key
      mPrefix       [403] OCTET STRING OPTIONAL,
      -- PrefixBlock encrypted with sender key
      cPrefix       [404] OCTET STRING OPTIONAL,
      -- HeaderBlock encrypted with sender key
      header        [405] OCTET STRING OPTIONAL,
      -- RoutingBlock encrypted with sender key
      routing       [406] OCTET STRING OPTIONAL,
      -- contains information for building messages (when used as MURB
      -- ID 0 denotes original message; ID 1-maxPayloadBlocks denotes
      -- target message; 32768-maxWorkspaceId shared workspace for all
      -- blocks of this identity)
      assembly      [407] SEQUENCE (SIZE (0..maxAssemblyInstructions))
                    OF PayloadOperation,
      validity      [408] UsagePeriod,
      -- optional - to identify the sender of a message when received
      id            [409] INTEGER OPTIONAL
    }

    PuzzleIdentifier          ::= OCTET STRING ( SIZE(0..16) )

    ChainSecret ::= INTEGER (0..4294967295)

  END
```

A.2. The VortexMessage Operation Structures

```
  MessageVortex-Operations DEFINITIONS EXPLICIT TAGS ::=
  BEGIN
    EXPORTS PayloadOperation;
    IMPORTS maxID
                  FROM MessageVortex-Schema
            SymmetricKey, AsymmetricKey, MacAlgorithmSpec, CipherSpec
                  FROM MessageVortex-Ciphers
            UsagePeriod, BlendingSpec
                  FROM MessageVortex-Helpers;
```

```
-- maximum omega of the Galois field used
maxGFSize                 INTEGER ::= 16
-- maximum size of a message chunk
maxChunkSize              INTEGER ::= 4294967295
-- minimum size of a message chunk (should be -maxChunkSize)
minChunkSize              INTEGER ::= -4294967295

PayloadOperation ::= CHOICE {
  splitPayload    [150] SplitPayloadOperation,
  mergePayload    [160] MergePayloadOperation,
  encryptPayload  [300] EncryptPayloadOperation,
  decryptPayload  [310] DecryptPayloadOperation,
  addRedundancy   [400] AddRedundancyOperation,
  removeRedundancy[410] RemoveRedundancyOperation
}

PercentSizeBlock ::= SEQUENCE {
  fromPercent         INTEGER (0..100),
  toPercent           INTEGER (0..100)
}

AbsoluteSizeBlock ::= SEQUENCE {
  fromAbsolute        INTEGER (0..maxChunkSize),
  toAbsolute          INTEGER (minChunkSize..maxChunkSize)
  -- negative toAbsolute denominates end in absolute value
  -- specified from the end of block
}

SizeBlock ::= SEQUENCE{
  size CHOICE {
    percent     [15001] PercentSizeBlock,
    absolute    [15101] AbsoluteSizeBlock
  }
}

AddRedundancyOperation ::= SEQUENCE {
  inputId            [16000] INTEGER (0..maxID),
  dataStripes        [16001] INTEGER (1..254),
  redundancy         [16002] INTEGER (1..254),
  keys               [16003] SEQUENCE (SIZE (2..512))
                             OF SymmetricKey,
  outputId           [16004] INTEGER (1..maxID),
  gfSize             [16005] INTEGER (2..maxGFSize)
}

RemoveRedundancyOperation ::= SEQUENCE {
  inputId            [16000] INTEGER (0..maxID),
  dataStripes        [16001] INTEGER (1..254),
```

```
      redundancy         [16002] INTEGER (1..254),
      keys               [16003] SEQUENCE (SIZE (2..512))
                                 OF SymmetricKey,
      outputId           [16004] INTEGER (1..maxID),
      gfSize             [16005] INTEGER (2..maxGFSize)
    }

    SplitPayloadOperation ::= SEQUENCE {
      originalId         INTEGER (0..maxID),
      firstSize          SizeBlock,
      newFirstId         INTEGER (1..maxID),
      newSecondId        INTEGER (1..maxID)
    }

    MergePayloadOperation ::= SEQUENCE {
      originalFirstId      INTEGER (0..maxID),
      originalSecondId     INTEGER (0..maxID),
      newId                INTEGER (1..maxID)
    }

    EncryptPayloadOperation ::= SEQUENCE {
      originalId            INTEGER (0..maxID),
      key                   SymmetricKey,
      newId                 INTEGER (1..maxID)
    }

    DecryptPayloadOperation ::= SEQUENCE {
      originalId            INTEGER (0..maxID),
      key                   SymmetricKey,
      newId                 INTEGER (1..maxID)
    }

  END
```

A.3. The VortexMessage Ciphers Structures

```
  MessageVortex-Ciphers DEFINITIONS EXPLICIT TAGS ::=
  BEGIN
    EXPORTS SymmetricKey, AsymmetricKey, MacAlgorithmSpec, CipherSpec;

    CipherSpec ::= SEQUENCE {
        asymmetric [16001] AsymmetricAlgorithmSpec OPTIONAL,
        symmetric  [16002] SymmetricAlgorithmSpec OPTIONAL,
        mac        [16003] MacAlgorithmSpec OPTIONAL,
        cipherUsage[16004] CipherUsage
    }

    CipherUsage ::= ENUMERATED {
```

```
    sign     (200),
    encrypt  (210)
  }

  SymmetricAlgorithmSpec  ::=  SEQUENCE  {
    algorithm               [16101]SymmetricAlgorithm,
    -- if ommited: pkcs1
    padding                 [16102]CipherPadding OPTIONAL,
    -- if ommited: cbc
    mode                    [16103]CipherMode OPTIONAL,
    parameter               [16104]AlgorithmParameters OPTIONAL
  }

  AsymmetricAlgorithmSpec  ::=  SEQUENCE  {
    algorithm               AsymmetricAlgorithm,
    parameter               AlgorithmParameters OPTIONAL
  }

  MacAlgorithmSpec  ::=  SEQUENCE  {
    algorithm               MacAlgorithm,
    parameter               AlgorithmParameters
  }

  PRNGAlgorithmSpec ::= SEQUENCE {
    type                    PRNGType,
    seed                    OCTET STRING
  }

  PRNGType ::= ENUMERATED {
    xsadd           (1000),
    blumMicali      (1001)
  }

  SymmetricAlgorithm ::= ENUMERATED {
    aes128          (1000),  -- required
    aes192          (1001),  -- optional support
    aes256          (1002),  -- required
    camellia128     (1100),  -- required
    camellia192     (1101),  -- optional support
    camellia256     (1102),  -- required
    twofish128      (1200),  -- optional support
    twofish192      (1201),  -- optional support
    twofish256      (1202)   -- optional support
  }

  CipherMode ::= ENUMERATED {
    -- ECB is a really bad choice. Do not use unless really
    -- necessary
```

```
      ecb             (10000),
      cbc             (10001),
      eax             (10002),
      ctr             (10003),
      ccm             (10004),
      gcm             (10005),
      ocb             (10006),
      ofb             (10007),
      none            (10100)
    }

    CipherPadding ::= ENUMERATED {
      none            (1000),
      pkcs1           (1001),
      pkcs7           (1002)
    }

    AsymmetricAlgorithm ::= ENUMERATED {
      rsa             (2000),
      dsa             (2100),
      ec              (2200),
      ntru            (2300)
    }

    MacAlgorithm ::= ENUMERATED {
      sha256          (3000),
      sha384          (3001),
      sha512          (3002),
      ripemd160       (3100),
      ripemd256       (3101),
      ripemd320       (3102)
    }

    ECCurveType ::= ENUMERATED{
      secp384r1       (2500),
      sect409k1       (2501),
      secp521r1       (2502)
    }

    AlgorithmParameters ::= SEQUENCE {
      keySize              [10000] INTEGER (0..65535) OPTIONAL,
      curveType            [10001] ECCurveType   OPTIONAL,
      initialisationVector [10002] OCTET STRING  OPTIONAL,
      nonce                [10003] OCTET STRING  OPTIONAL,
      mode                 [10004] CipherMode    OPTIONAL,
      padding              [10005] CipherPadding OPTIONAL,
      n                    [10010] INTEGER       OPTIONAL,
      p                    [10011] INTEGER       OPTIONAL,
```

```
      q                    [10012] INTEGER       OPTIONAL,
      k                    [10013] INTEGER       OPTIONAL,
      t                    [10014] INTEGER       OPTIONAL
    }

    -- Symmetric key
    SymmetricKey ::= SEQUENCE {
      keyType SymmetricAlgorithm,
      parameter AlgorithmParameters,
      key     OCTET STRING (SIZE(16..512))
    }

    -- Asymmetric Key
    AsymmetricKey ::= SEQUENCE {
      keyType       AsymmetricAlgorithm,
          -- private key encoded as PKCS#8/PrivateKeyInfo
      publicKey     [2] OCTET STRING,
          -- private key encoded as X.509/SubjectPublicKeyInfo
      privateKey    [3] OCTET STRING OPTIONAL
    }

  END
```

A.4. The VortexMessage Request Structures

```
  MessageVortex-Requests DEFINITIONS EXPLICIT TAGS ::=
  BEGIN
    IMPORTS StatusBlock, ReplyNodes, ReplyCurrentQuota, ReplyCapability
                  FROM MessageVortex-Replies
            RequirementBlock
                  FROM MessageVortex-Requirements
            UsagePeriod
                  FROM MessageVortex-Helpers;

    HeaderRequest ::= CHOICE {
      identity      [0] HeaderRequestIdentity,
      capabilities  [1] HeaderRequestCapability,
      messageQuota  [2] HeaderRequestIncreaseMessageQuota,
      transferQuota [3] HeaderRequestIncreaseTransferQuota,
      quotaQuery    [4] HeaderRequestQuota,
      nodeQuery     [5] HeaderRequestNodes
    }

    SpecialBlock ::= CHOICE {
      capabilities  [1]  ReplyCapability,
      requirement   [2]  SEQUENCE (SIZE (1..127))
                         OF RequirementBlock,
      quota         [4]  ReplyCurrentQuota,
```

```
      nodes         [5]  ReplyNodes,
      status        [99] StatusBlock
    }

    HeaderRequestIdentity ::= SEQUENCE {
      period UsagePeriod
    }

    HeaderRequestQuota ::= SEQUENCE {
    }

    HeaderRequestNodes ::= SEQUENCE {
      numberOfNodes   INTEGER
    }

    HeaderRequestIncreaseMessageQuota ::= SEQUENCE {
      messages INTEGER (0..4294967295)
    }

    HeaderRequestIncreaseTransferQuota ::= SEQUENCE {
      size     INTEGER (0..4294967295)
    }

    HeaderRequestCapability ::= SEQUENCE {
      period UsagePeriod
    }


  END
```

A.5. The VortexMessage Replies Structures

```
  MessageVortex-Replies DEFINITIONS EXPLICIT TAGS ::=
  BEGIN
    EXPORTS StatusBlock, ReplyCurrentQuota, ReplyCapability,
            ReplyNodes;
    IMPORTS BlendingSpec, NodeSpec
                  FROM MessageVortex-Helpers
            CipherSpec
                  FROM MessageVortex-Ciphers;

    StatusBlock ::= SEQUENCE {
      code            StatusCode
    }

    StatusCode ::= ENUMERATED {

      -- System messages
```

```
      ok                          (2000),
      quotaStatus                 (2101),
      puzzleRequired              (2201),

      -- protocol usage failures
      transferQuotaExceeded       (3001),
      messageQuotaExceeded        (3002),
      requestedQuotaOutOfBand     (3003),
      identityUnknown             (3101),
      messageChunkMissing         (3201),
      messageLifeExpired          (3202),
      puzzleUnknown               (3301),

      -- capability errors
      macAlgorithmUnknown         (3801),
      symmetricAlgorithmUnknown   (3802),
      asymmetricAlgorithmUnknown  (3803),
      prngAlgorithmUnknown        (3804),
      missingParameters           (3820),
      badParameters               (3821),

      -- Mayor host specific errors
      hostError                   (5001)
    }

    ReplyNodes ::= SEQUENCE {
        node   SEQUENCE (SIZE (1..5))
               OF NodeSpec
    }

    ReplyCapability ::= SEQUENCE {
      cipher            SEQUENCE (SIZE (2..256)) OF CipherSpec,
      maxTransferQuota  INTEGER (0..4294967295),
      maxMessageQuota   INTEGER (0..4294967295),
      supportedBlendingIn SEQUENCE OF BlendingSpec
    }

    ReplyCurrentQuota ::= SEQUENCE {
      messages INTEGER (0..4294967295),
      size     INTEGER (0..4294967295)
    }

  END
```

A.6. The VortexMessage Requirements Structures

```
MessageVortex-Requirements DEFINITIONS EXPLICIT TAGS ::=
BEGIN
  EXPORTS RequirementBlock;
  IMPORTS MacAlgorithmSpec
              FROM MessageVortex-Ciphers
          UsagePeriod, UsagePeriod
              FROM MessageVortex-Helpers;

  RequirementBlock ::= CHOICE {
    puzzle  [1] RequirementPuzzleRequired,
    payment [2] RequirementPaymentRequired
  }

  RequirementPuzzleRequired ::= SEQUENCE {
    -- bit sequence at beginning of hash from encrypted identity
    -- block
    challenge     BIT STRING,
    mac           MacAlgorithmSpec,
    valid         UsagePeriod,
    identifier    INTEGER (0..4294967295)
  }

  RequirementPaymentRequired ::= SEQUENCE {
    account       OCTET STRING,
    ammount       REAL,
    currency      Currency
  }

  Currency ::= ENUMERATED {
    btc      (8001),
    eth      (8002),
    zec      (8003)
  }

END
```

A.7. The VortexMessage Helpers Structures

```
MessageVortex-Helpers DEFINITIONS EXPLICIT TAGS ::=
BEGIN
  EXPORTS UsagePeriod, BlendingSpec, NodeSpec;
  IMPORTS AsymmetricKey, SymmetricKey
                FROM MessageVortex-Ciphers;

  -- the maximum number of parameters that might be embedded
  maxNumberOfParameter      INTEGER ::= 127

  UsagePeriod ::= SEQUENCE {
    notBefore       [0]     GeneralizedTime OPTIONAL,
    notAfter        [1]     GeneralizedTime OPTIONAL
  }

  -- contains a node spec of a routing point
  -- At the moment either smtp:<email> or xmpp:<jabber>
  BlendingSpec ::= SEQUENCE {
    target            [1] NodeSpec,
    blendingType      [2] IA5String,
    parameter         [3] SEQUENCE ( SIZE (0..maxNumberOfParameter) )
                          OF BlendingParameter
  }

  BlendingParameter ::= CHOICE {
    offset          [1] INTEGER,
    symmetricKey    [2] SymmetricKey,
    asymmetricKey   [3] AsymmetricKey
  }

  NodeSpec ::= SEQUENCE {
    transportProtocol [1] Protocol,
    recipientAddress  [2] IA5String,
    recipientKey      [3] AsymmetricKey OPTIONAL
  }

  Protocol ::= ENUMERATED {
    smtp  (100),
    xmmp  (110)
  }

END
```

A.8. The VortexMessage Additional Structures

```
-- States: Tuple()=Value() [vallidity; allowed operations] {Store}
-- - Tuple(identity)=Value(messageQuota,transferQuota,sequence of
--   Routingblocks for Error Message Routing) [validity; Requested
--   at creation; may be extended upon request] {identityStore}
```

```
   -- - Tuple(Identity,Serial)=maxReplays ['valid' from Identity
   --   Block; from First Identity Block; may only be reduced]
   --   {IdentityReplayStore}

   MessageVortex-NonProtocolBlocks DEFINITIONS EXPLICIT TAGS ::=
   BEGIN
     IMPORTS PrefixBlock, InnerMessageBlock, RoutingBlock, maxID
                  FROM MessageVortex-Schema
             UsagePeriod, NodeSpec, BlendingSpec
                  FROM MessageVortex-Helpers
             AsymmetricKey
                  FROM MessageVortex-Ciphers
             RequirementBlock
                  FROM MessageVortex-Requirements;

     -- maximum size of transfer quota in bytes of an identity
     maxTransferQuota          INTEGER ::= 4294967295
     -- maximum size of message quota in messages of an identity
     maxMessageQuota           INTEGER ::= 4294967295

     -- do not use these blocks for protocol encoding (internal only)
     VortexMessage ::= SEQUENCE {
       prefix        CHOICE {
           plain           [10011] PrefixBlock,
           -- contains prefix encrypted with receivers public key
           encrypted       [10012] OCTET STRING
       },
       innerMessage  CHOICE {
           plain     [10021] InnerMessageBlock,
           -- contains inner message encrypted with Symmetric key from
           -- Prefix
           encrypted [10022] OCTET STRING
       }
     }

     MemoryPayloadChunk ::= SEQUENCE {
       id                        INTEGER (0..maxID),
       payload                   [100] OCTET STRING,
       validity                  UsagePeriod
     }

     IdentityStore ::= SEQUENCE {
       identities SEQUENCE (SIZE (0..4294967295))
                  OF IdentityStoreBlock
     }

     IdentityStoreBlock ::= SEQUENCE {
       valid               UsagePeriod,
```

```
      messageQuota        INTEGER (0..maxMessageQuota),
      transferQuota       INTEGER (0..maxTransferQuota),
      -- if omitted this is a node identity
      identity            [1001] AsymmetricKey OPTIONAL,
      -- if ommited own identity key
      nodeAddress         [1002] NodeSpec      OPTIONAL,
      -- Contains the identity of the owning node;
      -- May be ommited if local node
      nodeKey             [1003] SEQUENCE OF AsymmetricKey OPTIONAL,
      routingBlocks       [1004] SEQUENCE OF RoutingBlock OPTIONAL,
      replayStore         [1005] IdentityReplayStore,
      requirement         [1006] RequirementBlock OPTIONAL
    }

    IdentityReplayStore ::= SEQUENCE {
      replays   SEQUENCE (SIZE (0..4294967295))
                OF IdentityReplayBlock
    }

    IdentityReplayBlock ::= SEQUENCE {
      identity            AsymmetricKey,
      valid               UsagePeriod,
      replaysRemaining  INTEGER (0..4294967295)
    }

  END
```

Author's Address

Martin Gwerder
Untere Parkstrasse 9
Hausen, AG 5212
Switzerland

Phone: +41 56 202 76 81
Email: rfc@messagevortex.net