J. Ylitalo
V. Torvinen
Ericsson Research Nomadiclab
E. Nordmark
Sun Microsystems, Inc.

Internet-Draft
Expires: July 28, 2004
January 28, 2004

Weak Identifier Multihoming Protocol (WIMP)
draft-ylitalo-multi6-wimp-00

Status of this Memo

Copyright Notice

Abstract

This draft defines a Weak Identifier Multihoming Protocol (WIMP) for IPv6 hosts. WIMP uses a new logical layer between networking and transport layers that separates the end-point identifier and locator roles from each other. The identifiers are used to name the end-points, while IPv6 addresses are used for routing purposes. WIMP is based on opportunistic security principles, and it does not require any pre-configured security infrastructure. The protocol consists of context establishment and re-addressing exchanges. The context establishment exchange creates a state for both hosts. The re-addressing exchange is used to update locator information at the

peer host. Used cryptographic operations are light and efficient.

Table of Contents

1. Introduction

The goal of the IPv6 multihoming work is to allow a site to take advantage of multiple attachments to the global Internet without having a specific entry for the site visible in the global routing table. Specifically, a solution should allow users to use multiple attachments in parallel, or to switch between these attachment points dynamically without an impact on the upper layer protocols.

This draft defines a Weak Identifier Multihoming Protocol (WIMP) for IPv6 hosts. WIMP uses reverse hash chains and secret splitting to perform enough validation to prevent redirection attacks.

The goals for WIMP are:

- Have no impact on upper layer protocols in general and on transport protocols in particular.
- Address the security threats in [6].
- Allow routers rewriting the (source) locators as a means of quickly detecting which locator is likely to work for return traffic.
- Minimal per-packet overhead.
- No extra round trip for setup through optional piggy-backing.

o Take advantage of multiple locators for load spreading.

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. Terminology

Upper Layer Protocol (ULP)

A protocol layer immediately above IP. Examples are transport protocols such as TCP and UDP, control protocols such as ICMP, routing protocols such as OSPF, and internet or lower-layer protocols being "tunneled" over (i.e., encapsulated in) IP such as IPX, AppleTalk, or IP itself.

Interface

A node's attachment to a link.

Locator

An IP layer topological name for an interface. The 128 bit locators are carried in the IP address fields as the packets traverse the network.

Identifier

An IP layer identifier for an IP layer endpoint (stack name in [7]). The transport endpoint is a function of the transport protocol and would typically include the IP identifier plus a port number.

Address field

The source and destination address fields in the IPv6 header. This draft separates identifiers from locators, and therefore these fields contain locators.

FQDN

Fully Qualified Domain Name

2.1 Cryptographic techniques

2.1.1 Reverse hash chain

Reverse hash chain is cryptographically generated list of data entities with some interesting characteristics. It is practically impossible to calculate or otherwise figure out the next value in the chain even when you know the previous value. However, it is very easy to verify whether some given value is the next value or not.

The chain is created by recursively computing a hash function over a result of the same function. The initial argument for the first hash value computation is typically a large random number. The last generated value of the chain is called as the "anchor" or "root" value. The hash values are revealed in the reverse order starting from the anchor value.

```
Hn = Hash(challenge)
Hn-1 = Hash(Hn)
...
H0 = anchor = Hash(H1)

CHAIN:  H0,..,Hn
```

This technique is usually applied based on an assumption that only an authentic end-point knows the correct successor values of the chain. In the bootstrapping process, the end-point computes a new hash chain and reveals the anchor value of the chain to its peer.

Each hash chain can be used only once.

2.1.2 Reverse hash chain and message authentication

Hashed message authentication codes (HMAC; RFC2104) are typically used to authenticate messages with a symmetric key. This requires, naturally, that all communicating peers share a secret.

Example: HMAC(key, Message).

Reverse hash chain values can be used as keys in the message authentication. This is different from the shared secret scheme because anybody who is able to receive the subsequent messages is able to verify that the messages belong together. The reverse hash chain value (key) used in message authentication is not revealed before the next message. In other words, the peer is able verify the message only after it has received the next message. This procedure can be used to verify that all subsequent messages come from the same entity than the first message. In other words, the hash chain binds the messages together.

A Man-in-the-Middle (MitM) attacker is not able to (unnoticeably) modify the messages after the first message in the exchange. However, an attacker may spoof the first message and use own hash chain. The protocol is based on opportunistic principle where the first message must be sent by an authentic node. The last message should contain only the last hash value, Hn.


```
Message_1: Message, HMAC(H1, Message)
Message_2: Message, H1, HMAC(H2, Message)
Message_3: Message, H2, HMAC(H3, Message)
...
Message_n: Message, Hn-1, HMAC(Hn, Message)
Message_n+1: Hn
```

2.1.3 Chained bootstrapping

In the basic model, each reverse hash chain is an independent entity. This is not a problem if the anchor of the chain can be authenticated, and if the hash chain is known to be long enough to have values to every message in the communication session. However, it is possible to use short hash chains to avoid extra computation.

Basically, the bootstrapping message can be authenticated using public keys. In the opportunistic mode, the peers do not authenticate the bootstrapping message with signatures, but rely that all subsequent messages are coming from the same peer. Two independently created reverse hash chains can be linked together with a Hash computation. The last value of the first reverse hash chain is used to authenticate the first value of the second chain:

```
...
Message_n: Message, H1_new, Hn-1, HMAC(Hn, Message)
Message_n+1: Message, Hn, HMAC(H2_new, Message)
Message_n+2: Message, H2_new, HMAC(H3_new, Message)
...
```

2.1.4 Secret splitting

In secret splitting, the secret is divided into several pieces. Any piece alone does not give enough information for an attacker to create the original secret. The only way to create the secret is to posses all the pieces of the key.

The simplest scheme splits a secret into two pieces. The splitting is done by generating a random-bit string, the same length as the secret, that will represent one piece of the split. The secret is then XORed with the random string to generate the other piece. Essentially, the secret is encrypted with a one-time pad. The pieces are the encrypted message, and the pad.

Example:

```
secret XOR random_string = encrypted_message

secret_piece_1 = random_string
secret_piece_2 = encrypted_message
```

The original secret can be reconstructed by XORing the pieces together. However, each piece alone can not be used to reconstruct the secret, not even with the highest possible computing power. One secret can very easily be divided into more than two pieces. What is needed is just more random-bit strings that are XORed into the secret.

Secret splitting is useful technique for storing secrets in two physical places, or for sending a secret to the other end-point using two or more parallel communication paths.

2.2 Notational Conventions

- I is an initiating host, i.e. an initiator.

- R is a responding host, i.e. a responder.

- ID(I) is an identifier for I.

- ID(R) is an identifier for R.

- FQDN(R) is the domain name for R.

- Ls(I) is the locator set for I, which consists of L1(I), L2(I), ... Ln(I).

- Ls(R) is the locator set for R.

- F(I) is a flowid assigned by the initiator and used by the responder. The responder uses F(I) in packets going to the initiator. F(I) is a tag for the host-pair context at the initiator.

- F(R) is a flowid assigned by the responder and used by the initiator.

- Hk(I) is k:th hash value in a initiator's reverse hash chain. The 'H0(I)' is the first revealed value, i.e. the "anchor" of the reverse hash chain. Note that the parameter k defines the revealing order, not the computation order.

- Hk(R) is k:th hash value in a responder's reverse hash chain.

3. Protocol overview

In order to prevent redirection attacks WIMP relies on the ability to verify that the entity requesting redirection indeed holds the successor values of a hash chain and is able to combine a divided secret that is sent via parallel paths. WIMP can be divided into two exchanges: context establishment and re-addressing exchange. The former exchange establishes a state for both communication end-points. The re-addressing exchange is used to update the locators belonging to the communicating parties. WIMP layer hides multiple addresses from the upper layer protocols.

3.1 WIMP layer

WIMP layer is located between IP and the ULPs, as shown in figure 1, in order to provide ULP independence. Conceptually the WIMP layer behaves as if it is an extension header, which would be ordered immediately after any hop-by-hop options in the packet.

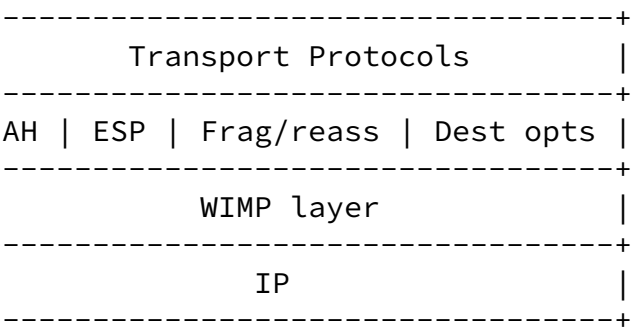


Figure 1: Protocol stack

All protocols above WIMP layer use 128 bit identifiers (IDs). The IDs are non-routable by their nature and never used in the IPv6 header in the network. An ID consists of an initial 2 bit prefix of 01, followed by 126 bits suffix that is a hash. The suffix in the initiator's identifier, ID(I), is a hash of a nonce. The suffix in

the responder's identifier, ID(R), is constructed taking a hash of the responder's FQDN(R).

Applications and upper layer protocols use IDs which the WIMP layer will map to/from locators (see figure 2). The WIMP layer maintains state, called host-pair context, in order to perform this mapping. The mapping is performed consistently at the initiator and the responder, thus from the perspective of the upper layer protocols packets appear to be sent using IDs from end to end, even though the packets travel through the network containing locators in the IP address fields, and even though those locators might be rewritten in flight. The result of this consistent mapping is that there is no

impact on the ULPs. In particular, there is no impact on pseudo-header checksums and connection identification.

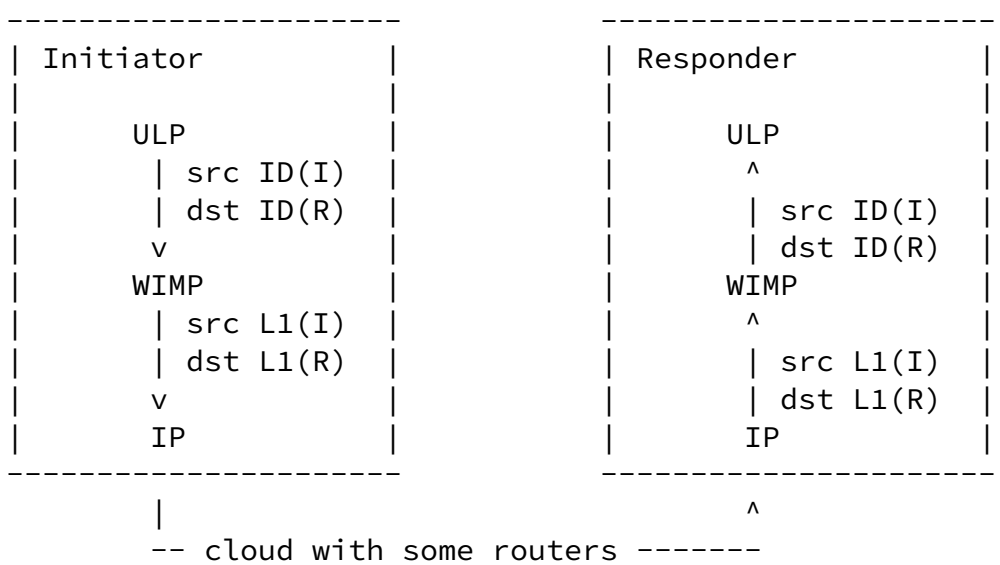


Figure 2: Mapping identifiers to locators.

Layering AH and ESP above the WIMP means that IPsec can be made to be unaware of locator changes the same way that transport protocols can be unaware. Thus the IPsec security associations remain stable even though the locators are changing. Layering the fragmentation header above the WIMP makes reassembly robust in the case that there is broken multi-path routing which results in using different paths, hence potentially different source locators, for different fragments.

3.2 Host-Pair Context

The initiator creates a host-pair context based on IDs and the locator set learned from the DNS. The responder establishes a state after receiving the second message from the initiator.

The context state contains the following information:

- Context identifiers (e.g. identities and flow id's)

- Locator and locator status (e.g. if locator has been verified, and which locators are preferred for communication)

- Hash chain information (e.g. parameters needed in the construction of hash chains, last used local chains values, and last known peer chain values)

Every IP packet must contain information about the related host-pair context. Basically, every packet could contain an extra extension

header including the IDs. However, that would add extra overhead to the packets. Instead, we use flowid field in the IPv6 header to represent the host-pair context. Conceptually one could view the approach as if both IDs being present in every packet, but with a header compression mechanism applied that removes the need for the IDs once the state has been established. The flowid serves as a convenient "compression tag" without increasing the packet size.

Flowids are used in regular IPv6 packets to find the right context for received packets. Each host selects for itself the flowid it wants to see in packets received from its peer. This allows it to select different flowids for different peers.

The problem related to flowid selection is identical with the IPSEC SPI selection. A host cannot select a flowid for IP packets going to its peer. Otherwise, two different hosts might select the same flowid and the peer could not uniquely identify the correct host-pair context using the received flowid.

Selected flowid is communicated to the peer in the third (CCR) and fourth (CONF) packets of the context creation exchange.

3.3 Generating reverse hash chains

In general, the way by which the reverse hash chains are generated is a local matter. The hash chains are needed by the initiator and the responder during the context establishment exchange. Also, the initiator of the re-addressing exchange needs to construct a new hash chain.

This document sets the following requirements for the hash chains generation:

- Each reverse hash chain MUST be unique for each host-pair context establishment and re-addressing exchange.

- The responder MUST be able to reconstruct the same reverse hash chain based on the parameters that are present in the context check reply message (CCR) in order to be stateless at the beginning of context establishment exchange.

- The responder MUST be able to reconstruct the same reverse hash chain based on the parameters that are present in the context check reply message (CCR) even if the system boots.

- The initial argument of the hash chain is never revealed to other parties.

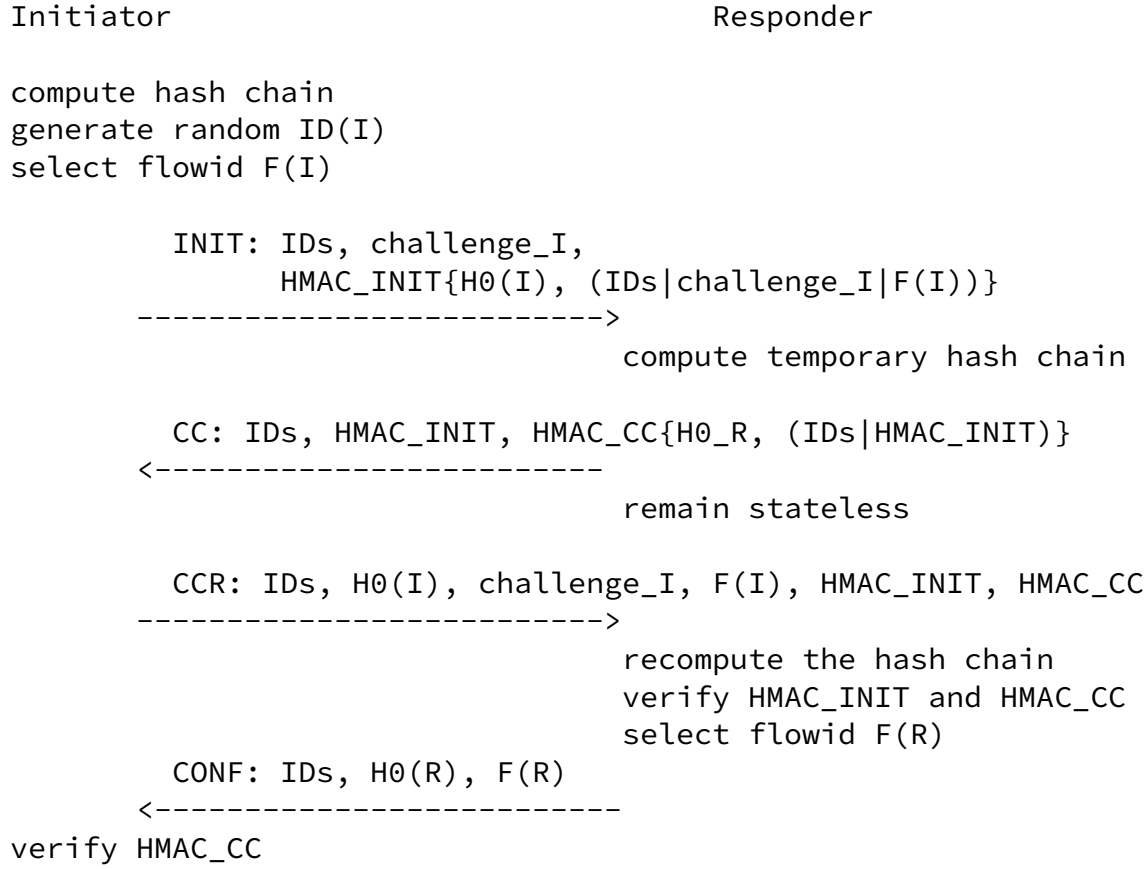
- The length of the hash chain is implementation specific but must be the same every time a chain is constructed.

Theoretically speaking, the minimum length of the hash chain is three (3) hash values. This will last for one context establishment exchange, and one re-addressing exchange for both direction (note that re-addressing exchange includes a bootstrapping procedure where a new hash chain is created for the initiator). However, each unsuccessful re-addressing exchange attempt will require one more hash chain value. Failures in re-addressing exchange may be due to connection loss in some of the locators, or an active attack.

Appendix A gives one example of secure hash chain generation.

3.4 Context establishment exchange

The context establishment exchange serves to manage the establishment of state between an initiator and a responder. The procedure uses delayed authentication principle where initial message exchange are verified with the parameters included in the latter messages. The procedure is designed to be stateless for the responder side. At the end of the exchange both parties have a uniquely identifiable host-pair context containing the peer's anchor value.



The initiator first sends a context initiator message, INIT, to the responder. The INIT message contains the IDs, a challenge and a keyed hash. The hash, having the anchor value as a key, is computed over the IDs, challenge and flowid. The flowid is sent later in the context check reply message (CCR).

Once the responder receives the INIT message, it must check that it does not already have a host-pair context with the ID pair. If the

context is not found, the responder continues with the negotiation. However, it does not want to establish a state because it is not able to verify the origin of the message. In order to remain stateless, the responder computes a temporary hash chain using the initiator's challenge in the INIT packet, and sends a context check message (CC) to the initiator. The CC message is protected with the anchor value of responder hash chain.

The initiator replies to the CC message with a context check reply message (CCR) and proves that it was reachable at the specific location by disclosing the anchor value. The CCR message includes the flowid that uniquely identifies the host pair context to the initiator.

Again, the responder does not accept CCR packets with an ID pair that already has a host pair-context. If the context is not found, the responder recomputes its own hash chain and verifies the keyed hashes (HMAC_INIT and HMAC_CC). The anchor value of the initiator hash chain binds the INIT and CCR messages together, and in this way the responder is able to verify that messages are coming from the same host. If the keyed hashes were valid, the responder names the host pair context by generating a flowid for it, and replies with a context confirmation message (CONF) revealing its anchor value.

The initiator verifies the keyed hash in the CC message with the anchor value received in the CONF message, and finalizes its state.

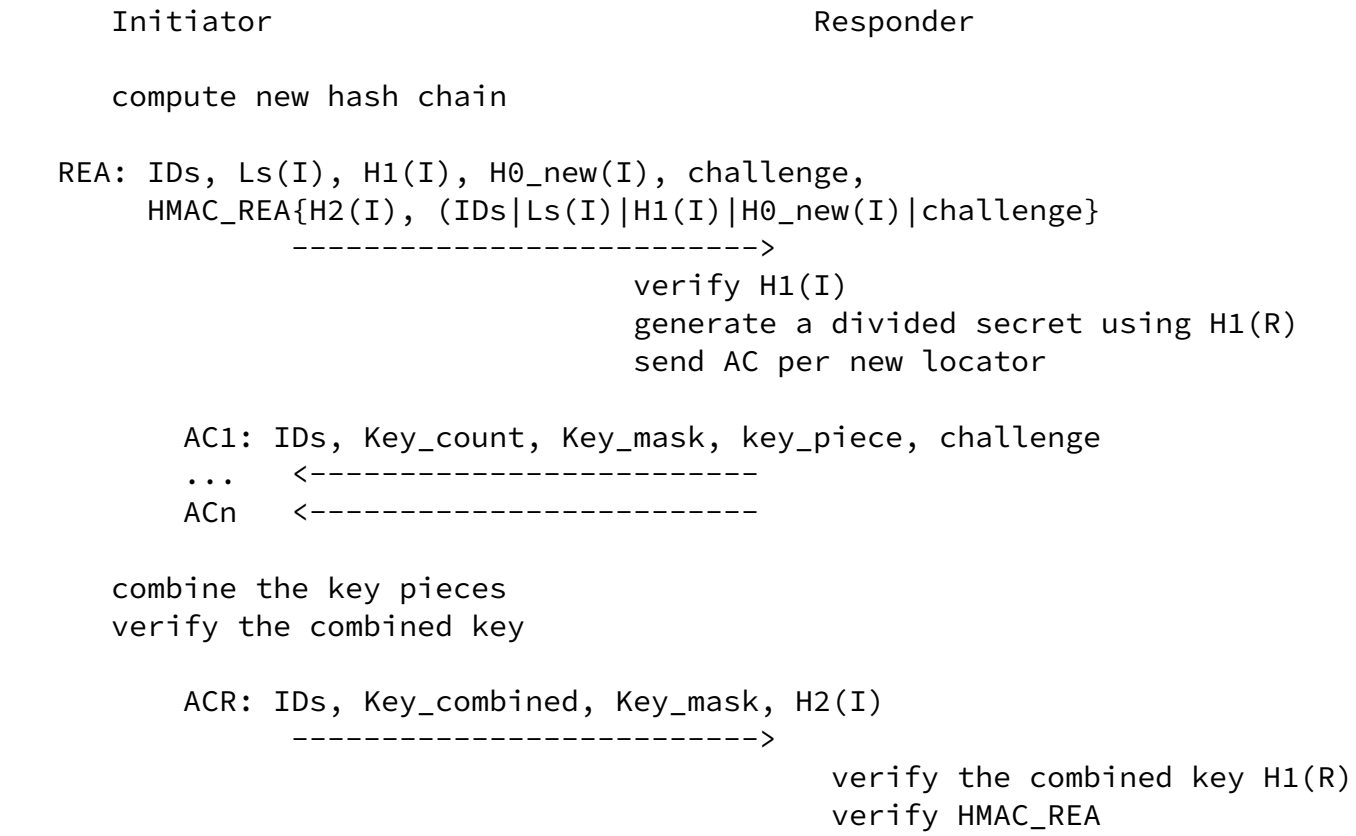
The context establishment exchange has been designed in the way that it can be reused for secure resynchronisation. If the initiator receives a RESYNC message, it SHOULD start a new handshake with the INIT message by including the latest disclosed responder's hash chain value. In this way, the initiator can be sure that the RESYNC message really originated from the responder, and not from an attacker. In this case, the responder discloses the successor hash chain value in the CONF message, instead of the anchor value.

3.5 Re-addressing exchange

Once the state has been completed, both hosts have a host-pair context. At this point, the initiator has received the responder's

locator set from the DNS. However, it is possible that the responder

has published only one IP address in the DNS, but it is still able to multihome, and has several IP addresses. Basically, both hosts may send to their peers the locator sets. A host may verify the peer's addresses on demand basis or all at once.



The re-addressing exchange is a three way handshake. The re-addressing message (REA) has two purposes. First, it contains the initiator's locator set. Second, it bootstraps a new hash chain.

Once the responder receives a REA message, it verifies that the hash chain value H1(I) belongs to the initiator (this example assumes that the previously revealed hash chain value was the anchor, H0). In addition, the responder stores the initiator's new anchor value H0_new(I) and the new challenge value into the host-pair context.

The responder divides its own next unused hash chain value into pieces using the secret splitting technique. The hash chain value is divided into as many parts as there were locators in the received locator set. The responder defines a key mask for each of the key pieces (see Section 6.12). The key mask will later allow the responder to check if all pieces found their way to the initiator. It is possible that the initiator is not reachable via all of the locators in the locator set.

The responder sends one address check message (AC) per locator in the received locator set. Each AC message contains one piece of the hash chain value. The initiator must have a local policy on how long it waits for the upcoming AC messages.

If the initiator receives all pieces of the hash chain value, it verifies that it is a successor value of the responder's hash chain. Otherwise, the initiator MAY stop the re-addressing exchange procedure. The initiator makes an OR operation with all of the received key masks and includes the result of the operation with the combined key to the address check reply message (ACR).

The responder verifies the combined key using the key mask that indicates the key pieces that the initiator was able to receive. In addition, the responder uses the received initiator's hash chain value to authenticate the earlier received REA message. Finally, the responder marks the corresponding locators in the initiator's locator set as verified.

4. Packet processing

Each host is assumed to have a separate WIMP implementation that manages the host's host-pair context and handles requests for new ones. Each host-pair context is governed by a state machine, with states defined in Section 7. WIMP implementation can simultaneously maintain host-pair contexts with more than one host. Furthermore, WIMP implementation may have more than one active host-pair context with another host; in this case, host-pair contexts are distinguished by their respective IDs and flowids. It is not possible to have more than one host-pair contexts between any given pair of IDs. Consequently, the only way for two hosts to have more than one parallel associations is to use different IDs, at least in one end.

The processing of packets depends on the state of the host-pair context(s) with respect to the originator of the packet. A WIMP implementation determines whether it has an active association with the originator of the packet based on the IDs or the flowid of the packet.

4.1 Processing outgoing application data

In an WIMP host, an application can send application level data using IDs as source and destination identifiers. The IDs can be specified via a backwards compatible API. However, whenever there is such outgoing data, the stack has to send the resulting datagram using appropriate source and destination IP addresses.

The following steps define the conceptual processing rules for outgoing datagrams destinated to a ID(R).

1. If the datagram has a specified source address, it MAY be a ID(I). If it is not, the implementation MAY replace the source address with a ID(I). Otherwise it MUST send the packet using the given IP addresses and bypass the WIMP layer.

2. If the datagram has an unspecified source address, the implementation MUST choose a suitable source ID(I) for the datagram. In selecting a proper ID(I), the implementation MUST

consult the table of currently active WIMP sessions, and preferably select a ID(I) that already has an active session with the target ID(R).

3. If there is no active WIMP session with the ID(R), one must be created by running the context establishment exchange. The implementation MAY piggy-pack ULP payload to the exchange packets.

4. Once there is an active WIMP session for the given ID(R), the outgoing datagram does not contain WIMP headers. Instead, the IDs are represented by flowids in the IP headers.

5. The IDs in the datagram are replaced with suitable IP addresses. The rules defined in [2] MAY be followed.

4.2 Processing incoming application data

Incoming WIMP datagrams arrive as normal IP packets containing WIMP flowids. In the usual case the receiving host has a corresponding host-pair context, identified by the flowid in the packet. However, if the host has crashed or otherwise lost its WIMP state, it may not have such a host-pair context.

The following steps define the conceptual processing rules for incoming datagrams targeted to a WIMP host-pair context.

1. Detect the proper host-pair context, using the flowid. If there are no host-pair context identified with the flowid, the host MAY reply with a RESYNC packet. Sending such RESYNCs MUST be rate limited. However, if the receiver has an open socket corresponding the IP addresses in the IP header it MUST process the packet as standard IP datagram.

2. If a proper host-pair context is found, the packet is processed by WIMP. The IP addresses in the datagram are replaced with the IDs associated with the host-pair context.

3. The datagram is delivered to the upper layer. Demultiplexing the datagram the right upper layer socket is based on the IDs.

5. Packets

There are eight basic WIMP packets. Four are for the context establishment exchange, three are for re-addressing, and one is for state loss.

Packets consist of the fixed header as described in Section 6.1, followed by the parameters. The parameter part, in turn, consists of zero or more TLV coded parameters.

Packet representation uses the following operations:

()	parameter
[]	optional parameter

An upper layer payload MAY follow the WIMP header. The payload proto field in the header indicates if there is additional data following the WIMP header. The WIMP packet, however, MUST NOT be fragmented. This limits the size of the possible additional data in the packet.

5.1 INIT - the context initiator packet

The WIMP header values for the INIT packet:

Header:

```
      Packet Type = 1
      SRC ID = Initiator's ID
      DST ID = Responder's ID

   IP( WIMP ( CHALLENGE, HASH-INIT ))
```

Valid control bits: P

The INIT packet contains the fixed WIMP header, initiator's challenge, and HASH-INIT TLV (Section 6.3).

If the INIT message is a response to a RESYNC packet the initiator SHOULD replace the challenge with the responder's old challenge. In this way, the initiator is able to verify later that the responder has really lost its state.

The Initiator generates the Responder's identifier, ID(R), using the the Responder's FQDN. If the Initiator does not know the Responder's FQDN, it MUST use IP addresses for the application identifiers in the socket API and by pass the WIMP protocol.

5.2 CC - the context check packet

The WIMP header values for the CC packet:

```
   Header:
     Packet Type = 2
     SRC ID = Responder's ID
     DST ID = Initiator's ID

   IP ( WIMP ( HASH-INIT, HASH-CC ))
```

Valid control bits: P

The IDs MUST be the same received in INIT. (If the Responder has multiple IDs, the responder ID used MUST match Initiator's request.)

The Responder copies the HASH-INIT TLV from INIT message to the CC message

The Responder computes a temporary hash chain using the received initiator's challenge and IDs (see Section 3.3). The responder uses the generated anchor value of the temporary hash chain as a key for the HASH-CC computation Section 6.4). (Note: The responder does not store the temporary hash chain values.)

5.3 CCR - the context check reply packet

The WIMP header values for the CCR packet:

```
Header:
  Type = 3
  SRC ID = Initiator's ID
  DST ID = Responder's ID

IP ( WIMP ( ANCHOR, FLOWID, CHALLENGE, [HASHVAL,] HASH-INIT, HASH-CC )
```

Valid control bits: P

The IDs used MUST match the ones used in the previous messages.

If the initiator already has a host-pair context for the responder, but responder has lost its state, the initiator SHOULD sent the latest known responder's hash value, HASHVAL, and the old challenge, to the responder.

The Initiator reveals the anchor value that was used in HASHKEY TLV in the HASH-INIT computation in the INIT message.

Initiator copies the HASH-INIT, and HASH-CC TLVs from the CC message to the CCR message. In addition, CCR message includes the anchor, flowid, challenge, and responder's latest hash value. .The Initiator selects a flowid that the responder uses to send packets to the Initiator (Section 3.2)

5.4 CONF - the context confirm packet

The WIMP header values for the CONF packet:

```
Header:
```

```
      Packet Type = 4
      SRC ID = Responder's ID
      DST ID = Initiator's ID

   IP ( WIMP ( ANCHOR, FLOWID ))
```

Valid control bits: P

The Responder reveals the anchor value that was used in HASHKEY TLV in the HASH-CC computation in the CC message.

The Responder selects a flowid that the initiator uses to send packets to the Responder.

5.5 RESYNCH - the re-synchronization packet

The WIMP header values for the RESYNC packet:

```
   Header:
     Packet Type = 5
     SRC ID = Responder's ID
     DST ID = (zero)

   IP ( WIMP ( FLOWID ))
```

Valid control bits: -

Once a responder receives an IP packet with an unknown flowid it replies with RESYNC packet and copies the received flowid into the message. However, if the responder has an open socket corresponding the IP addresses in the IP header it MUST process the packet as a standard IP datagram.

5.6 REA - The re-addressing packet

The WIMP header values for the REA packet:

```
Header:
  Packet Type = 10
  SRC ID = Initiator's ID
  DST ID = Responder's ID

IP ( WIMP ( LSET, HASHVAL, ANCHOR, CHALLENGE, HASH-REA ))
```

Valid control bits: P

The REA message contains initiator's locator set. The node that was the responder during the context establishment exchange is the initiator when it initiates a re-addressing exchange. The initiating party of re-addressing exchange is called the initiator in the REA context.

The Initiator revels a successor hash value (HASHVAL TLV) of its current hash chain. In addition, it bootstraps a new hash chain by revealing a new anchor value (ANCHOR TLV). The challenge used to generate a new hash chain is included into the message. Hash chain generation for the initiator is discussed in Section Section 3.3.

Section 6.5 describes how the HASH-REA is computed.

5.7 AC - The address check packet

The WIMP header values for the AC packet:

```
Header:
  Packet Type = 11
  SRC ID = Responder's ID
  DST ID = Initiator's ID

IP ( WIMP ( KEY, CHALLENGE ))
```

Valid control bits: P

The responder divided a secret into as many pieces as there were addresses in locator set in the received REA message. The key count contains the total number of the key pieces. The key mask is an identifier for a key piece in the KEY TLV. The challenge TLV includes the challenge received in the REA message. The initiator is able to bind the received AC to the correct exchange using the challenge. The

responder sends one AC packet per locator in the received locator set. (Note: Each AC contains one key piece.)

5.8 ACR - The address check reply packet

The WIMP header values for the AC packet:

```
Header:
  Packet Type = 12
  SRC ID = Initiator's ID
  DST ID = Responder's ID

IP ( WIMP ( KEY, HASHVAL ))
```

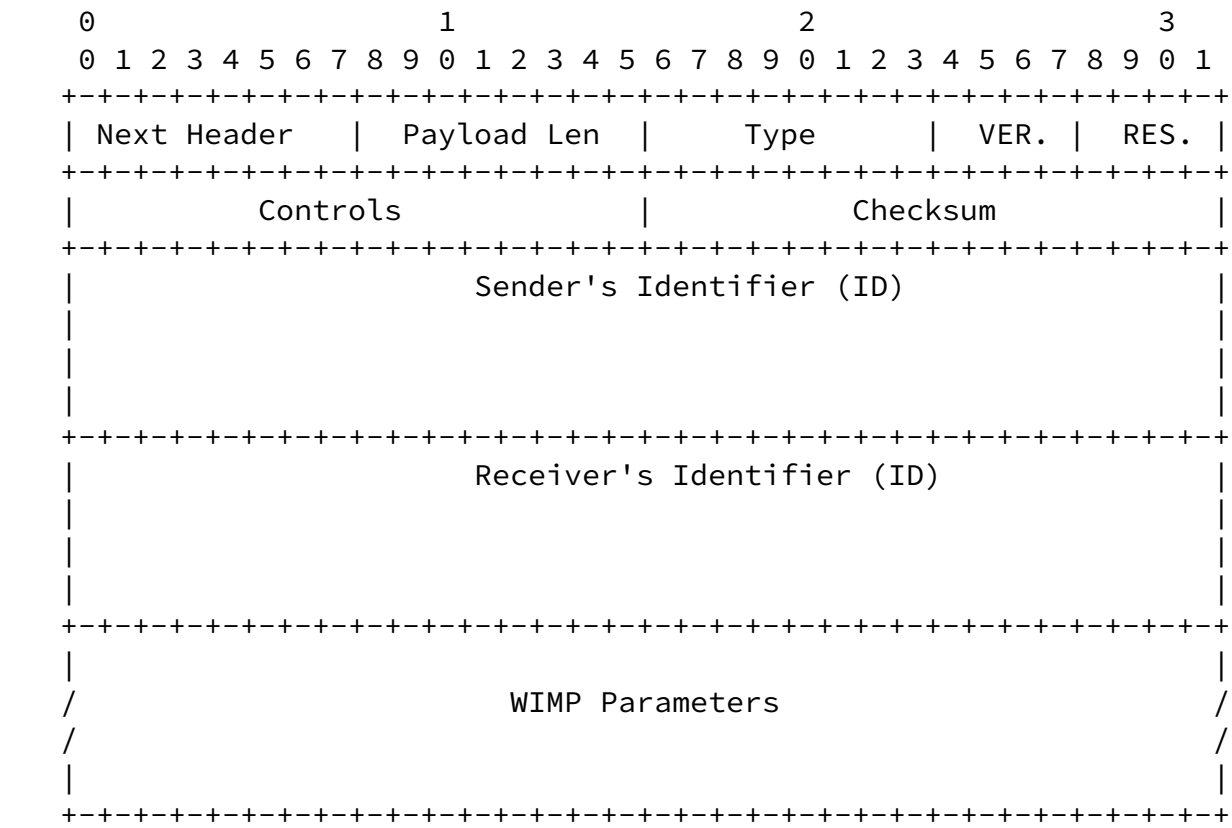
Valid control bits: P

The KEY TLV is constructed of the received key pieces and their key masks. The HASHVAL TLV contains the hash value that was used as a key in the REA packet in HASH-REA TLV.

6. Message formats

6.1 Payload format

All WIMP packets start with a fixed header.



The WIMP header is logically an IPv6 extension header followed by a next header that is defined in Next Header field. If no next headers follow, the decimal 59, IPPROTO_NONE, the IPV6 no next header value, is used.

The Header Length field contains the length of the WIMP header and the length of parameters in 8 bytes units, excluding the first 8 bytes. Since all WIMP headers MUST contain the sender's and receiver's ID fields, the minimum value for this field is 4, and

conversely, the maximum length of the WIMP Parameters field is (255*8)-32 = 2008 bytes.

The Packet Type indicates the WIMP packet type. The individual packet types are defined in the relevant sections. If a WIMP host receives a packet that contains an unknown packet type, it MUST drop the packet.

The Version is four bits. The current version is 1. The version number is expected to be incremented only if there are incompatible changes to the protocol. Most extensions can be handled by defining new packet types, new parameter types, or new controls.

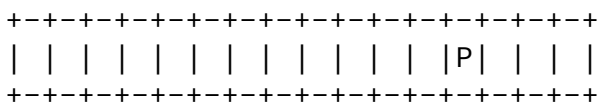
The following four bits are reserved for future use. They MUST be zero when sent, and they SHOULD be ignored when handling a received packet.

The ID fields are always 128 bits (16 bytes) long.

6.1.1 WIMP Controls

The WIMP control section transfers information about the structure of the packet and capabilities of the host.

The following fields have been defined:



P - Piggy backing The sending host is capable of sending and receiving additional data (e.g. ESP) in WIMP packets.

The rest of the fields are reserved for future use and MUST be set to zero on sent packets and ignored on received packets.

6.1.2 Checksum

The checksum field is located at the same location within the header as the checksum field in UDP packets, enabling hardware assisted checksum generation and verification. Note that since the checksum

covers the source and destination addresses in the IP header, it must be recomputed on WIMP based middle-boxes.

The pseudo-header [3] contains the source and destination IPv6 addresses, WIMP packet length in the pseudo-header length field, a zero field, and the WIMP protocol number (TBD) in the Next Header field. The length field is in bytes and can be calculated from the WIMP header length field: (WIMP Header Length + 1) * 8.

6.2 Parameters

The parameters are used to carry the hash chain values associated with the sender's ID, together with other related security information. The parameters consists of ordered parameters, encoded in TLV format.

The following parameter types are currently defined.

TLV	Type	Length	Data
(TBD)			

6.2.1 TLV format

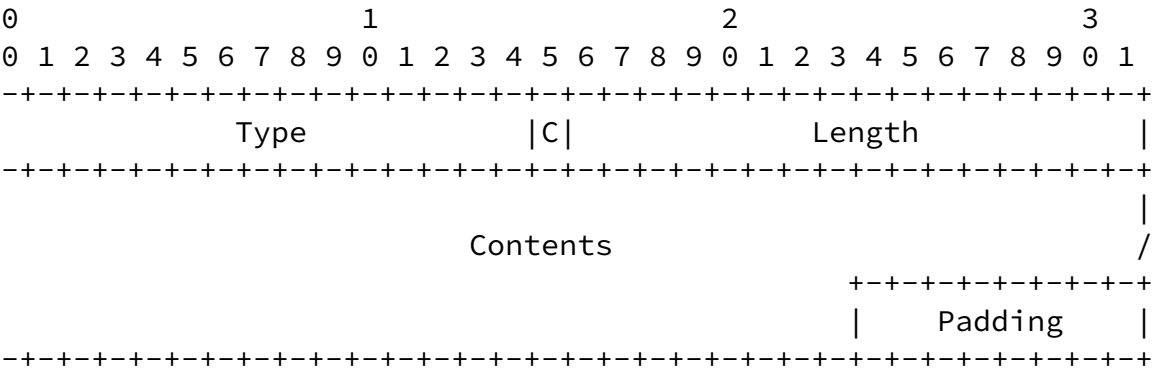
The TLV encoded parameters are described in the following subsections. The type-field value also describes the order of these fields in the packet. The parameters MUST be included into the packet so that the types form an increasing order. If the order does not follow this rule, the packet is considered to be malformed and it MUST be discarded.

All the TLV parameters have a length which is a multiple of 8 bytes. When needed, padding MUST be added to the end of the parameter so that the total length becomes a multiple of 8 bytes. This rule ensures proper alignment of data. If padding is added, the Length field MUST NOT include the padding. Any added padding bytes MUST be set zero by the sender, but their content SHOULD NOT be checked on the receiving end.

Consequently, the Length field indicates the length of the Contents

field (in bytes). The total length of the TLV parameter (including Type, Length, Contents, and Padding) is related to the Length field according to the following formula:

```
Total Length = 11 + Length - (Length + 3) % 8;
```

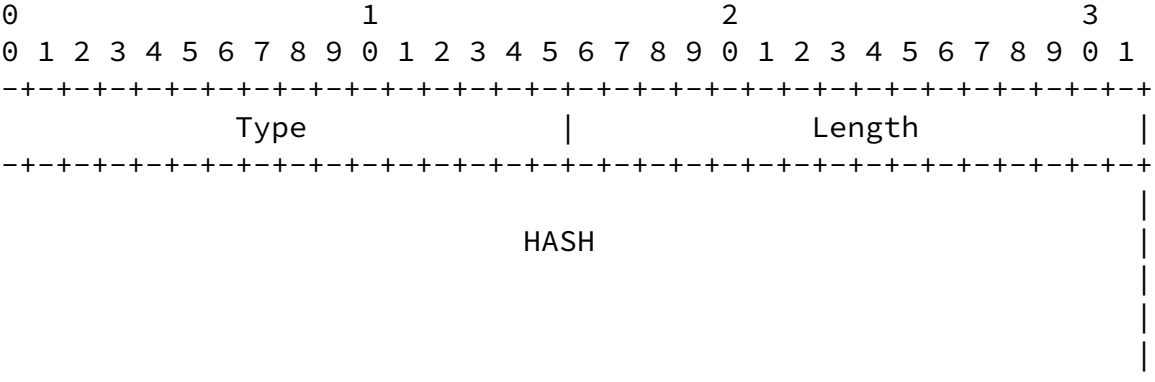
Type	Type code for the parameter
C	Critical. One if this parameter is critical, and MUST be recognized by the recipient, zero otherwise. The C bit is considered to be a part of the Type field. Consequently, critical parameters are always odd and non-critical ones have an even value.
Length	Length of the parameter, in bytes.
Contents	Parameter specific, defined by Type

Padding	Padding, 0-7 bytes, added if needed

Critical parameters MUST be recognized by the recipient. If a recipient encounters a critical parameter that it does not recognize, it MUST NOT process the packet any further.

Non-critical parameters MAY be safely ignored. If a recipient encounters a non-critical parameter that it does not recognize, it SHOULD proceed as if the parameter was not present in the received packet.

6.3 HASH-INIT



```
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

   Type            65500
   Length          20
   HASH

                   160 bit SHA1 is computed over:
                     - WIMP common header.
                     - HASHKEY TLV, Initiator's second hash value, H1(I)
                     - FLOWID TLV, initiator.
                     - CHALLENGE TLV, 128 challenge generated by the initiato
                   excluding the HASH-INIT parameter. The checksum field MUST
                   be set to zero and the WIMP header length in the WIMP comm
                   header MUST be calculated to cover all the included
                   parameters when the SHA1 is calculated.
```

HASH-INIT calculation and verification process:

INIT message sender:

1. Create the INIT message, containing HASHKEY, FLOWID, and CHALLENGE TLVs, without the HASH-INIT TLV.

2. Calculate the length field in the WIMP header.

3. Compute the SHA1.

4. remove the HASHKEY, and FLOWID TLVs.

5. Add the HASH-INIT TLV to the packet.

6. Recalculate the length field in the WIMP header.

CCR message receiver:

1. reconstruct the INIT message, containing HASHKEY, FLOWID, and

CHALLENGE TLVs, without the HASH-INIT TLV.

2. Calculate the WIMP packet length in the WIMP header and clear the checksum field (set it to all zeros).

3. Compute the HASH-INIT and verify it against the received HASH-INIT.

6.4 HASH-CC

The TLV structure is the same as in Section 6.3. The fields are:

Type	65502
Length	20
HASH	160 bit SHA1 is computed over: - WIMP common header. - HASHKEY TLV, the responder's hash chain value. - HASH-INIT TLV, received in the INIT message, excluding the HASH-CC parameter. The checksum field MUST be set to zero and the WIMP header length in the WIMP comm header MUST be calculated to cover all the included parameters when the SHA1 is calculated.

HASH-CC calculation and verification process:

CC message sender:

1. Create the CC message, containing HASHKEY, and HASH-INIT TLVs, without HASH-CC TLV.

2. Calculate the length field in the WIMP header.

3. Compute the SHA1.

4. remove the HASHKEY TLV.

5. Add the HASH-CC TLV to the packet.

6. Recalculate the length field in the WIMP header.

CCR and CONF message receiver:

1. reconstruct the CC message, containing HASHKEY, and HASH-INIT TLVs, without the HASH-CC TLV.

2. Calculate the WIMP packet length in the WIMP header and clear the

checksum field (set it to all zeros).

3. Compute the HASH-CC and verify it against the received HASH-CC.

6.5 HASH-REA

The TLV structure is the same as in Section 6.3. The fields are:

Type	65504
Length	20
HASH	160 bit SHA1 is computed over: - WIMP common header, - LSET TLV, Initiator's locator set. - HASHVAL TLV, a hash value, Hk, of the initiator's hash chain - HASHKEY TLV, a successor hash value, Hk+1, of the initiator's hash chain - ANCHOR TLV, the anchor value, H0, of the initiator's new hash chain - CHALLENGE TLV, the challenge used in the new hash chai generation. excluding the HASH-REA parameter. The checksum field MUST be set to zero and the WIMP header length in the WIMP comm header MUST be calculated to cover all the included parameters when the SHA1 is calculated.

HASH-REA calculation and verification process.

REA message sender:

1. Create the REA message, containing LSET, HASHVAL, ANCHOR, HASHKEY and CHALLENGE TLVs, without HASH-REA TLV.

2. Calculate the length field in the WIMP header.

3. Compute the SHA1.

4. remove the HASHKEY TLV.

5. Add the HASH-CC TLV to the packet.

6. Recalculate the length field in the WIMP header.

REA message receiver:

1. Add HASHKEY TLV

2. Remove and store HASH-REA TLV.

3. Calculate the WIMP packet length in the WIMP header and clear the checksum field (set it to all zeros).

4. Compute the HASH-REA and verify it against the received HASH-REA.

6.6 ANCHOR

The TLV structure is the same as in Section 6.3. The fields are:

Type	10
Length	20
HASH	160 bit SHA1. An anchor value of a hash chain.

6.7 HASHVAL

The TLV structure is the same as in Section 6.3. The fields are:

Type	12
Length	20
HASH	160 bit SHA1 hash value. It is generated by computing a hash over a successor hash chain value.

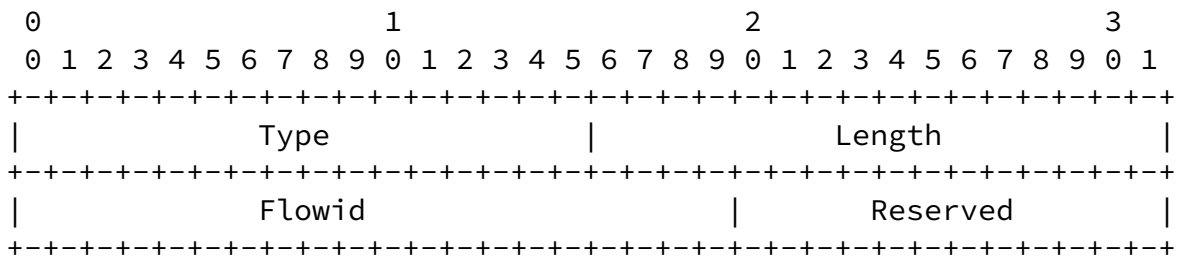
6.8 HASHKEY

The TLV structure is the same as in Section 6.3. The fields are:

Type	14
Length	20
HASH	160 bit SHA1 hash value that is used as a key in HASH-INIT, HASH-CC and HASH-REA TLV computation. The key value is never revealed in the same message as the corresponding authentication hash.

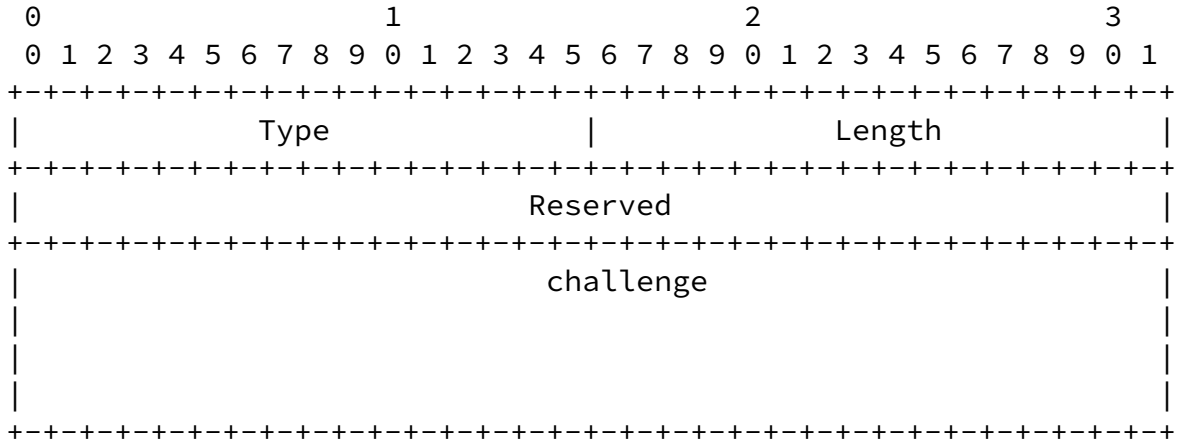
6.9 FLOWID

The FLOWID parameter contains the flowid that the receiving host must use when sending data to the sending host.



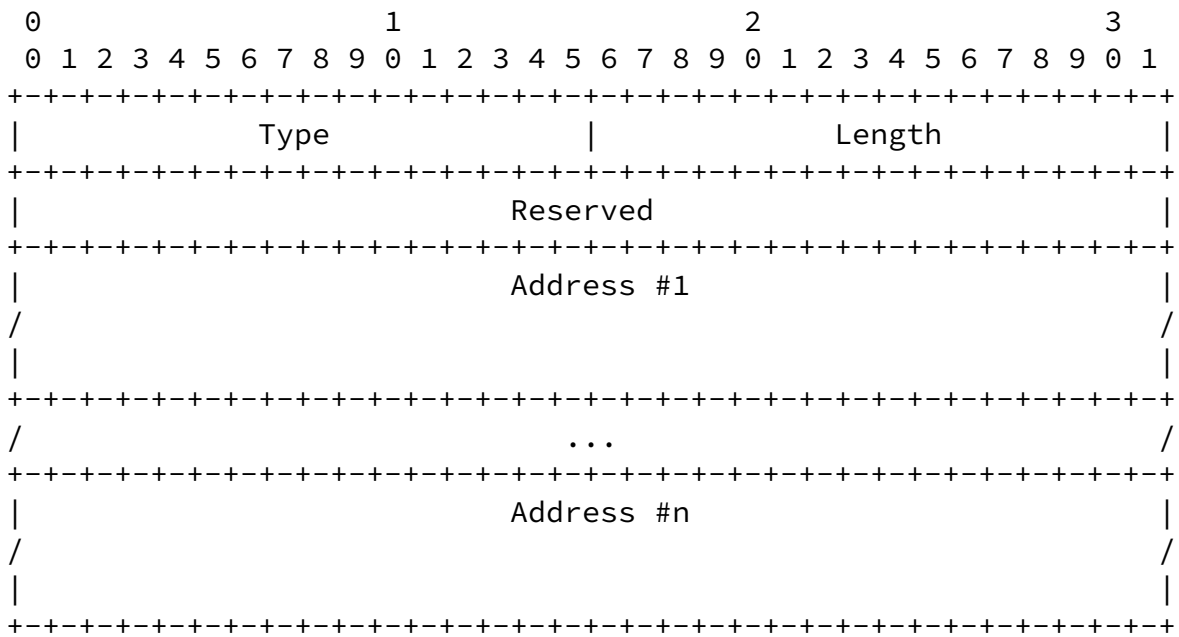
Type 16
Length 4
Reserved Zero when sent, ignored when received
Flowid 20 bit flowid

6.10 CHALLENGE



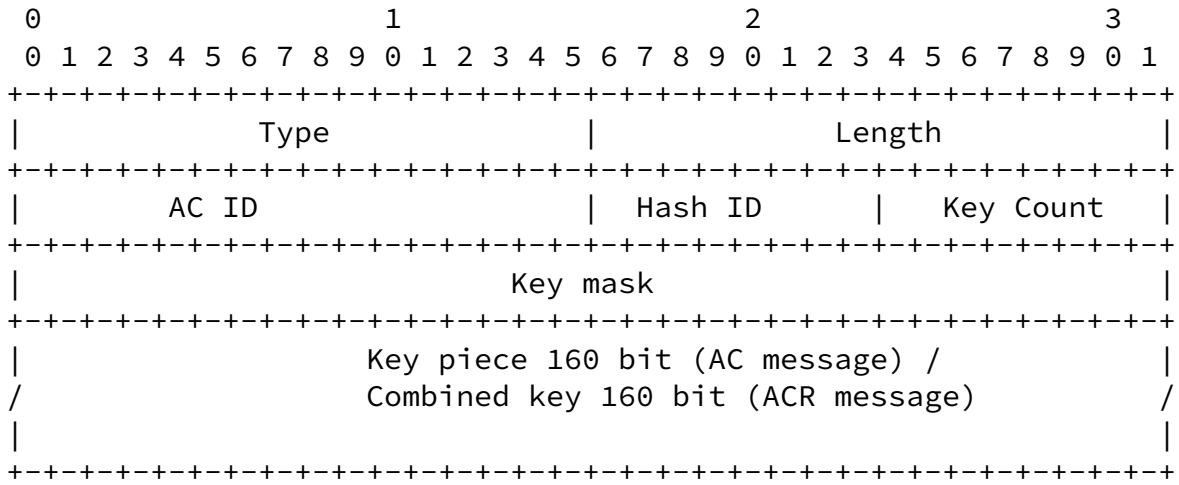
Type 20
Length 20
Reserved Zero when sent, ignored when received
challenge 128 bit random number

6.11 LSET



Type	22
Length	variable
Reserved	Zero when sent, ignored when received
Address	IPv6 address

6.12 KEY

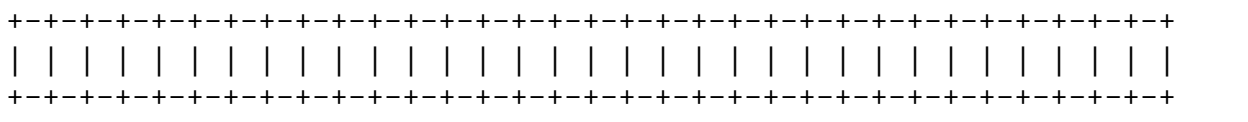


Type 26
Length variable
AC ID An increasing counter that identifies the exchange for the responder.
Hash ID The order number of a hash chain value in the responder's

hash chain. The hash chain value that is divided into key pieces.

Key count The total number of key pieces sent/received

Key Mask



The responder defines a bit in the mask for a key piece.

Key piece A hash chain value is divided into (160 bit) key pieces by the responder.

Combined key The initiator constructs a combined key using the key pieces.

The AC ID is related to a divided secret, i.e. a hash chain value. Every key piece related to that value must have the same AC ID.

Responder sending AC message:

The key count field value in the AC packet is the total number of key pieces sent by the responder. I.e. the total number of sent AC packets. Each KEY-MASK TLV in one AC packet contains a key piece and corresponding bit on in the key mask field

(Note: The responder is able to make a return routability test maximum for 32 addresses per divided key.)

Initiator sending ACR message:

The initiator makes an OR operation for all received key masks in the AC packets. The result of the operation is included in the KEY-MASK TLV in ACR message.The key count field in KEY-MASK TLV in the ACR message indicates the number of key pieces the initiator received from the responder. The combined key is included into the KEY-MASK TLV.

7. State Machine

The initiator of the context establishment exchange is called the initiator. Once the host-pair contexts are established, this initial distinction is lost. The sender of the REA message is called the initiator of the re-addressing exchange. In a case of state loss, the sender of the RESYNCH message becomes a responder. .

The state machine is presented in a single system view, representing either an Initiator or a Responder. There is not a complete overlap of processing logic here and in the packet definitions. Both are needed to completely implement WIMP.

Implementors must understand that the state machine, as described here, is informational. Specific implementations are free to implement the actual functions differently.

7.1 States

START , state machine start

INIT-sent , INIT sent by initiator

CCR-sent , CCR sent by initiator

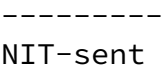
ESTABLISHED , host-pair context established

ESTABLISHED-REA-sent , REA sent

FAILED , host-pair context establishment failed

7.2 State Processes

Datagram to send, send INIT and go to INIT-send
Receive INIT, send CC and stay at START
Receive CCR, process
    if successful, send CONF and go to ESTABLISHED
    if fail, stay at START
Receive IP packet with unknown flowid, send RESYNC and stay at START
Receive ANYOTHER, drop and stay at START



```
+-----------+

Receive INIT, send CC and stay at INIT-sent
Receive CCR, process
     if successful, send CONF and go to ESTABLISHED
     if fail, stay at INIT-sent
Receive CC, process
     if successful, send CCR and go to CCR-sent
     if fail, stay at INIT-sent
Receive ANYOTHER, drop and stay at INIT-sent
Timeout, increment timeout counter
     If counter is less than INIT_RETRIES_MAX, send INIT and stay at INIT-se
     If counter is greater than INIT_RETRIES_MAX, go to FAILED

+-----------+
| CCR-sent |
+-----------+

Receive INIT, send CC and stay at CCR-sent
Receive CCR, process
     if successful, send CONF and go to ESTABLISHED
     if fail, stay at CCR-SENT
Receive CONF, process
     if successful, go to ESTABLISHED
     if fail, stay at CCR-SENT
Receive ANYOTHER, drop and stay at CCR-sent
Timeout, increment timeout counter
     If counter is less than CCR_RETRIES_MAX, send CCR and stay at CCR-sent
     If counter is greater than CCR_RETRIES_MAX, go to FAILED

+-------------+
| ESTABLISHED |
+-------------+

REA to send, go to ESTABLISHED-REA-sent
Receive RESYNC, send INIT and cycle at ESTABLISHED
Receive CC, process
     if successful, send CCR and stay at ESTABLISHED
     if fail, drop, and stay at ESTABLISHED
Receive CONF, process
     if successful, update host-pair context and cycle at ESTABLISHED
     if fail, drop, and stay at ESTABLISHED
Receive REA, process
     if successful, update host-pair context, send ACs,
        and cycle at ESTABLISHED
     if fail, drop, and stay at ESTABLISHED
Receive ACR, process
     if successful, update host-pair context, and cycle at ESTABLISHED
```

```
     if fail, drop, and stay at ESTABLISHED
Receive IP packet for host-pair context, process and stay at ESTABLISHED
Receive ANYOTHER, drop and stay at ESTABLISHED

+----------------------+
| ESTABLISHED-REA-sent  |
+----------------------+

Receive RESYNC, send INIT and go to ESTABLISHED
Receive REA, process
     if successful, update host-pair context, send ACs,
        and cycle at ESTABLISHED-REA-sent
     if fail, drop, and cycle at ESTABLISHED-REA-sent
Receive AC, process
     if successful, send ACR, and go to ESTABLISHED
     if fail, sent REA, and cycle at ESTABLISHED-REA-sent
Receive ACR, process
     if successful, update host-pair context,
        and cycle at ESTABLISHED-REA-sent
     if fail, drop, cycle at ESTABLISHED-REA-sent
Receive IP packet for host-pair context, process,
      and stay at ESTABLISHED-AC-sent
Receive ANYOTHER, drop and stay at ESTABLISHED-AC-sent
Timeout, increment timeout counter
     If counter is less than REA_RETRIES_MAX, send REA,
         and stay at ESTABLISHED-REA-sent
     If counter is greater than REA_RETRIES_MAX, go to FAILED
```

7.3 State Loss

WIMP protocol and state machine is designed to recover from one of the parties crashing and losing its state. The following scenarios describe the main use cases covered by the design.

No prior state between the two systems.

The system with data to send is the Initiator. The process follows standard 4 packet context establishment exchange, establishing the host-pair context.

The initiator has no state with the responder, but the responder has an earlier host-pair context.

Initiator acts as in no prior state, sending INIT. The Initiator selects a new random ID(I) and the responder finally establishes a new host-pair context with the initiator. The old context will expire due to timers (TBD).

The initiator has a host-pair context, but the responder does not.

The responder 'detects' the situation when it receives a packet that contains an unknown flowid. The responder sends a RESYNC with received flowid and a NULL (zero) initiator ID. The initiator gets the RESYNC and initiates a new context establishment exchange to re-establish the host-pair context with the existing IDs. The initiator generates a new hash chain but sends the old challenge, to the responder.

If a host reboots or times out, it has lost its host-pair context. If the system that lost state has a datagram to deliver to its peer, it simply restarts the context establishment exchange with INIT message containing a new ID(I). The responder replies with CC packet.

If a system receives an IP packet for an unknown flowid, the assumption is that it has lost the state and its peer did not. In this case, the system replies with a RESYNCH packet. The ID(I) is typically NULL in the RESYNCH, since the system usually does not know the peer's ID any more.

The system receiving the RESYNCH packet first checks to see if it has an established and recently used host-pair context with the party sending the RESYNCH. If such a context exists, the initiator initiates a new context establishment exhange by sending INIT message. Note that the process will result in a new host-pair context at the responder site, while the initiator is able to use the existing host-pair context.

8. Security Considerations

The initial design choice was to use reverse hash chains instead of public key technology. The reason for this was that there exists already a group of authenticated Diffie-Hellman key exhange protocols. However, protocols using public key technology are vulnerable for different kind of CPU related denial-of-service attacks. The trade-off between hash chain based message authentication and signatures is that hash chains are vulnerable for a class of man-in-the-middle attacks. However, if we accept that the first round-trip of the context establishment exchange is open for such attacks we obtain several advantages. The hash chain computation is a lightweight operation compared to signature verification.

8.1 Context establishment exchange

8.1.1 Man-in-the-Middle attacks

The context establishment exchange is based on opportunistic authentication. In other words, both hosts, the initiator and responder, have to trust the first message comes from the authentic peer. The responder trusts that the INIT message comes from an authentic initiator. In a similar way, the initiator trusts that the CC message comes from an authentic responder. Therefore, the first round-trip is vulnerable for the MitM attacks.

The second round-trip of the exchange bootstraps the hash chains and bounds the messages together. The anchor values revealed during the second round-trip are used to authenticate the first round-trip

messages. A MitM attacker cannot change any values in the CCR message, because the message is authenticated with the responder's HMAC. The responder's HMAC, in turn, is computed over the initiator's HMAC. In this way, the responder does not need to establish a state once it receives INIT message. In addition, a MitM attacker cannot reserve a host-pair context by sending CCR message with initiator's ID because the messages are bound together. Finally, the hash chains have two main purposes. They bind messages together, and they are used to authenticate the messages. The hash chain properties were discussed in more detail in Section 2.1.

8.1.2 Denial-of-Service attacks

The initiator may use any identifier, ID(I), it wants with the responder. This property protects the initiator from a specific form of DoS attack. That is, the attacker is not able to establish a context with the responder using the initiator's identifier if the initiator chooses its identifier randomly.

On the other hand, the beginning of the context establishment exchange is stateless for the responder in order to avoid attacks where the attacker is trying to create inconsistent states in the responder. The responder does a kind of return-routability test before establishing a state. The initiator has to prove that is reachable in the location where it sent the initial packet. The responder may optionally restrict establishing a host-pair context for an initiator if the responder already has several host-pair contexts related to same IP addresses.

Basically, the context establishment protocol is vulnerable for a flowid related attack. The only value that is not protected with HMAC is the responder's flowid in the last message. The reason for this is that the responder cannot reserve any values before it creates a state. The state is created after receiving the CCR message. As a consequence, the responder cannot include the flowid into its HMAC in the CC message. a MitM attacker may change the responder's flowid on the fly in the last message and course a denial-of-service situation. In other words, the initiator will send IP packets with wrong flowid to the responder. However, the main objective of the protocol was to protect the hosts from the re-direction attacks and the presented attack does not open new vulnerabilities for that part.

8.2 Re-addressing exchange

When a site renumbers, a host inside the site must inform its peers about the new IP address(es). The sender of the new locator set is called an initiator. Once the responder receives REA message it cannot trust that the initiator is reachable at the new locations. To avoid different kind of DoS and redirection attacks the responder must verify that the initiator indeed is reachable at the claimed locations.

the proposed protocol takes advantage of parallel paths between the hosts. The responder splits its hash chain value into pieces and protects each challenge (AC) message with one piece. Each of challenge messages is sent to different location. As a consequence, an attacker has to locate at different topological locations, at the same time, to be able to answer to the challenge. The initiator, in turn, is able to verify that all of the pieces came from the authentic responder. The secret splitting works only if there are more than one AC messages to be sent and the messages are routed via different paths to the initiator. (Note: To increase the security of the challenge message, the responder could protect the AC message with HMAC. The key in each message would then be a divided secret, i.e. a hash value.)

9. IANA Considerations

IANA has assigned IP Protocol number TBD to WIMP.

10. Acknowledgments

This draft is a result of the discussion between the authors, Pekka Nikander and Jari Arkko at the 58th IETF. The idea was to write a Multi6 draft using reverse hash chains and secret splitting. The main objective in this draft has been on identifying the hosts to their peers with reverse hash chains. Instead of inventing the wheel once again, we took several ideas from the existing protocol proposals.

Therefore, there are certain similarities with SIM and HIP design factors. We would like to thank all contributers in those drafts that have affect the work.

Normative references

[1] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

[2] Draves, R., "Default Address Selection for Internet Protocol version 6 (IPv6)", RFC 3484, February 2003.

[3] Deering, S. and R. Hinden, "Internet Protocol, Version 6 (IPv6) Specification", RFC 2460, December 1998.

[4] Abley, J., Black, B. and V. Gill, "Goals for IPv6 Site-Multihoming Architectures", RFC 3582, August 2003.

[5] Kent, S., "IP Encapsulating Security Payload (ESP)", draft-ietf-ipsec-esp-v3-05 (work in progress), April 2003.

[6] Nordmark, E. and T. Li, "Threats relating to IPv6 multihoming solutions", draft-nordmark-multi6-threats-00.txt (work in progress), October 2003.

[7] Lear, E. and R. Droms, "What's In A Name: Thoughts from the NSRG", draft-irtf-nsrg-report-09.txt (work in progress), March 2003.

Informative references

Authors' Addresses

Jukka Ylitalo
Ericsson Research Nomadiclab

Jorvas FIN-02420
FINLAND

Phone: +358 9 299 1
EMail: jukka.ylitalo@ericsson.com

Vesa Torvinen
Ericsson Research Nomadiclab

Turku FIN-20520
FINLAND

Phone: +358 9 299 1
EMail: vesa.torvinen@ericsson.com

Erik Nordmark
Sun Microsystems, Inc.
17 Network Circle
Mountain View, CA
USA

Phone: +1 650 786 2921
EMail: erik.nordmark@sun.com

Appendix A. Example of secure reverse hash chain generation

This Appendix gives an example for secure hash chain generation.

Initiator (context establishment and re-addressing exchanges):

```
Hn(I) = SHA1(secret(I)|ID(I)|ID(R)|challenge(I))
...
H1(I) = SHA1(H2(I))
H0(I) = SHA1(H1(I))
```

Responder (context establishment exchange):

```
Hn(R) = SHA1(secret(R)|ID(R)|ID(I)|challenge(I))
  Note: challenge(I) is generated by the initiator
...
H1(R) = SHA1(H2(R))
H0(R) = SHA1(H1(R))
```

The default length of both of the hash chains is n=10.

```
secret(I)    = 256 bit secret random number generated by the initiator
secret(R)    = 256 bit secret random number generated by the responder
challenge(I) = 128 bit public random number generated by the initiator
Hk(I/R)      = initiator's/responder's 160 bit hash chain value
ID(I/R)      = initiator's/responder's 128 bit identifier
```

The secret(I/R) is not revealed to the peers. The long term secret(R) MUST survive when the system boots. The responder SHOULD use the same secret(R) for different hash chains generated during the

lifetime of the secret(R).

Appendix B. Goals for IPv6 Site-Multihoming Architectures

This section compares the proposed solution with the goals of IPv6 site multihoming architecture [4].

B.1 Redundancy

The proposed end-to-end protocol allows applications to use multiple IPv6 addresses. Furthermore, the WIMP layer is able to receive events from different sources: from upper and lower layers. L2 may inform WIMP, e.g, if the local link is down. L4, in turn, may launch a trigger due to TCP time-outs. Finally, the redundancy is a question of local triggers, implementation and address selection policy. (The address selection policy should be separated from the core protocol draft)

B.2 Load Sharing

The identifier and locator-set separation makes it possible to implement dynamic load sharing. The sender is able to select source and destination IP addresses on packet basis. The sender and receiver may have several parallel paths between then; both of the hosts being multi-homed at the same time. (Note: However, sending TPC packets via alternative paths may lead to poor performance.)

B.3 Performance

The protocol defined in this draft does not directly interoperate with Internet routing protocols. However, the implementation should follow source address routing principles. In other words, the next hop router, and its prefix, defines the used source address. However, a host with one interface typically receives multiple prefixes from

one access router. Congestion related to a path should be identified using RTT and bandwidth values as triggers for hand-offs between addresses.

B.4 Policy

The proposed solution does not take a stand on the reason why multihoming is used. The solution provides support for site-multihoming for external policy reasons. If the responder has stored several IP addresses to the DNS, the initiator may assume that the responder supports multihoming. (Note: there are several other ways using DNS to inform the initiator that the peer supports multihoming.)

B.5 Simplicity

Proposed solution is straightforward to deploy and maintain. It is not more complex to deploy and operate than current IPv4 multihoming practices. The protocol supports legacy IPv6 socket API.

B.6 Transport-Layer Survivability

The proposed solution provides re-homing transparency for transport-layer sessions. The existing transport-layer sessions are able to use the new path because the upper layer identities are separated from the network topology.

B.7 Impact on DNS

No impacts on DNS RR records.

B.8 Packet Filtering

The solution does not preclude filtering.

B.9 Scalability

The proposed protocol is responsible for changing the default path, source and destination IP addresses, to another if the peer is not

reachable via some route. The routing desicion is made at the end-host.

B.10 Impact on Routers

The proposed solution does not require changes to IPv6 router implementations.

B.11 Impact on Hosts

A host that has not implemented the proposed solution can still work in a multihomed site. The solution requires changes to the end-host stack, however, these changes are logically separate functions that can be added to existing functions. The solution does not require changes to the socket API or the transport layer.

B.12 Interaction between Hosts and the Routing System

The solution does not require interaction between a site's hosts and its routing system.

B.13 Operations and Management

The staff responsible for the operation of a site is able to advertise any prefixes it wants for the hosts. The site multihoming system can be configured advertising different set of prefixes.

B.14 Cooperation between Transit Providers

The solution does not require cooperation between transit providers.

B.15 Security compared to IPv4 multihoming

The proposed solution is vulnerable for man-in-the-middle attack in the first message exchange because it relies on opportunistic security principles. The security properties are analyzed in Section 8.

Appendix C. Things MULTI6 Developers should think about

This section answers to the questions presented in the draft-lear-multi6-things-to-think-about-00.

C.1 Routing

"How will your solution solve the multihoming problem?"

By defininining a new logical layer between networking (L3) and transport (L4) layers. The logical layer separates identifiers from

locators. The dynamic one-to-many binding between an identifier and IP addresses makes multihoming possible. The identifiers are used by upper layers, while the IP addresses are used on the wire.

"Does your solution address mobility?"

The proposal does not define any rendevouz server, but allows dynamic address updates between hosts.

C.2 Identifiers and locators

"Does your solution provide for a split between identifiers and locators?"

Yes. The responder's identifier is a 128-bit hash of FQDN. The initiator's identifier, in turn, is a hash of a nonce for security reasons.

"What is the lifetime of a binding from an identifier to a locator?"

The lifetime of a binding is the lifetime of open sockets. As long as there are open connections between end-points, the corresponding host-pair context should not be deleted. However, the binding can be dynamically updated.

"How is the binding updated?"

By re-addressing exchange.

"Will transport connections remain up?"

Yes.

C.3 On The Wire

"At what layer is your solution applied, and how?"

At logical layer between networking (L3) and tranport (L4) layers.

"Is it applied in every packet? If so, what fields are used?"

It is applied in every packet by using flowid field in the IPv6 header

"Why is the layer you chose the correct one?"

The implementation does not affect the upper layers nor applications.

"Does your solution expand the size of an IP packet?"

Only during the context establishment and re-addressing exchanges. The IP payload packets use flowid field to identify packets.

"Do you change the way fragmenting is handled?"

Not in the normal IPv6 packets. The packets carrying context establishment or re-addressing exchange messages must not be fragmented.

"Are there any changes to ICMP error semantics?"

No. However, the presented packet formats can be implemented as ICMP extension fields.

C.4 Names, Hosts, Endpoints, or none of the above?

"Please explain the relationship of your solution to DNS"

The proposal assumes that a host is multi-homed if it has several AAAA RR entries in the DNS.

"Please explain interactions with "2-faced" DNS"

No effect

"Does your solution require centralized registration?"

Every server (responder) must have a basic DNS entry, containing FQDN and locator set, as nowadays

"Have you checked for DNS circular dependencies?"

The solution is dependent of FDQN and locator set mapping

"What if a DNS server itself is multihomed?"

No effect.

"What application/API changes are needed?"

None.

"Is this solution backward compatible with "old" IP version 6?"

Yes. The new identifiers are identified with two highest bits in the 128-bit identifier, like in the HIP.

"Is your solution backward compatible with IPv4?"

The 6to4 gateway must work as an authentication proxy. TBD.

"How will your solution interact with other middle-boxes?"

TBD.

"Are there any implications for scoped addressing?"

TBD.

"Are there any layer 2 implications to your proposal?"

None.

"How will your solution handle referrals, such as those within FTP?"

TBD.

"Are you introducing a namespace that might involve mnemonics?"

No.

Intellectual Property Statement

Full Copyright Statement

Acknowledgment

Funding for the RFC Editor function is currently provided by the Internet Society.