Network Working Group
Internet-Draft
Updates: 6126 (if approved)
Intended status: Experimental
Expires: October 20, 2013

D. Ovsienko
Yandex

April 18, 2013

Babel HMAC Cryptographic Authentication
draft-ovsienko-babel-hmac-authentication-03

Abstract

This document describes a cryptographic authentication mechanism for Babel routing protocol, updating, but not superceding RFC 6126. The mechanism allocates two new TLV types for the authentication data, uses HMAC and is both optional and backward compatible.

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 http://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 October 20, 2013.

Copyright Notice

Table of Contents

1. Introduction

[RFC Editor: before publication please remove the sentence below.] Comments are solicited and should be addressed to the author.

Authentication of routing protocol exchanges is a common mean of securing computer networks. Use of protocol authentication mechanisms helps in ascertaining that only the intended routers participate in routing information exchange, and that the exchanged routing information is not modified by a third party.

[BABEL] ("the original specification") defines data structures, encoding, and the operation of a basic Babel routing protocol instance ("instance of the original protocol"). This document ("this specification") defines data structures, encoding, and the operation of an extension to the Babel protocol, an authentication mechanism ("this mechanism"). Both the instance of the original protocol and this mechanism are mostly self-contained and interact only at coupling points defined in this specification.

A major design goal of this mechanism is transparency to operators that is not affected by implementation and configuration specifics. A complying implementation makes all meaningful details of authentication-specific processing clear to the operator, even when some of the key parameters cannot be changed.

The currently established (see [RIP2-AUTH], [OSPF2-AUTH], [OSPF3-AUTH], and [RFC6039]) approach to authentication mechanism design for datagram-based routing protocols such as Babel relies on two principal data items embedded into protocol packets, typically as two integral parts of a single data structure:

- A fixed-length unsigned integer, typically called a cryptographic sequence number, used in replay attack protection.

- A variable-length sequence of octets, a result of the HMAC construct (see [RFC2104]) computed on meaningful data items of the packet (including the cryptographic sequence number) on one hand and a secret key on the other, used in proving that both the sender and the receiver share the same secret key and that the meaningful data was not changed in transmission.

Depending on the design specifics either all protocol packets are authenticated or only those protecting the integrity of protocol exchange. This mechanism authenticates all protocol packets.

This specification defines the use of the cryptographic sequence number in details sufficient to make replay attack protection

strength predictable. That is, an operator can tell the strength from the declared characteristics of an implementation and, whereas the implementation allows to change relevant parameters, the effect of a reconfiguration.

This mechanism explicitly allows for multiple HMAC results per authenticated packet. Since meaningful data items of a given packet remain the same, each such HMAC result stands for a different secret key and/or a different hash algorithm. This enables a simultaneous, independent authentication within multiple domains.

An important concern addressed by this mechanism is limiting the amount of HMAC computations done per authenticated packet, independently for sending and receiving. Without these limits the number of computations per packet could be as high as the number of configured authentication keys (in the sending case) or as the number of keys multiplied by the number of supplied HMAC results (in the receiving case).

These limits establish a basic competition between the configured keys and (in the receiving case) an additional competition between the supplied HMAC results. This specification defines related data structures and procedures in a way to make such competition transparent and predictable for an operator.

Wherever this specification mentions the operator reading or changing a particular data structure, variable, parameter, or event counter "at runtime", it is up to the implementor how this is to be done. For example, the implementation can employ an interactive CLI, or a management protocol such as SNMP, or an inter-process communication mean such as a local socket, or a combination of these.

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 RFC 2119 [RFC2119].

2. Cryptographic Aspects

2.1. Mandatory-to-Implement and Optional Hash Algorithms

[RFC2104] defines HMAC as a construct that can use any cryptographic hash algorithm with a known digest length and internal block size. This specification preserves this property of HMAC by defining data processing that itself does not depend on any particular hash algorithm either. However, since this mechanism is a protocol

extension case, there are relevant design considerations to take into account.

Section 4.5 of [RFC6709] suggests selecting one hash algorithm as mandatory-to-implement for the purpose of global interoperability (Section 3.2 ibid.) and selecting another of distinct lineage as recommended for implementation for the purpose of cryptographic agility. This specification makes the latter property guaranteed, rather than probable, through an elevation of the requirement level. There are two hash algorithms mandatory-to-implement, unambiguously defined and generally available in multiple implementations each.

An implementation of this mechanism MUST include support for two hash algorithms:

- SHA-512 (SHA-2 family)
- Whirlpool 2nd ed., 2003 (512-bit hash)

Besides that, an implementation of this mechanism MAY include support for additional hash algorithms, provided each such algorithm is publicly and openly specified and its digest length is 16 octets or more (to meet the constraint set in Section 4.3). Implementors SHOULD consider strong, well-known hash algorithms as additional implementation options and MUST NOT consider hash algorithms for that by the time of implementation meaningful attacks exist or that are commonly viewed as deprecated. For example, the following hash algorithms meet these requirements at the time of this writing (in alphabetical order):

- GOST R 34.11-94 (256-bit hash)
- RIPEMD-160
- SHA-224 (SHA-2 family)
- SHA-256 (SHA-2 family)
- SHA-384 (SHA-2 family)
- Tiger (192-bit hash)

The set of hash algorithms available in an implementation MUST be clearly stated. When known weak authentication keys exist for a hash algorithm used in the HMAC construct, an implementation MUST deny a use of such keys.

2.2. Padding Constant Specifics

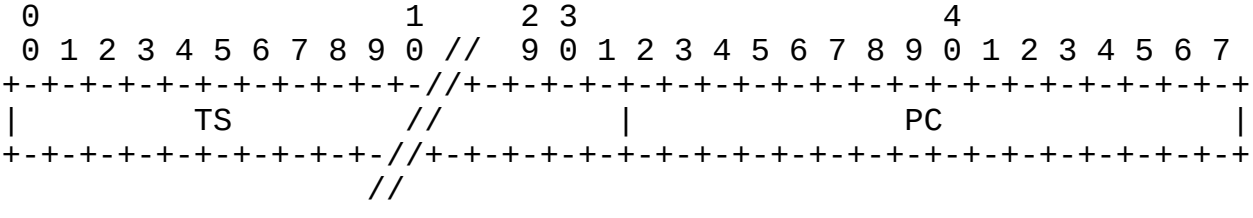
[RIP2-AUTH] established a reference method of routing protocol packets authentication using the HMAC construct. The method sets that a protocol packet being authenticated is sized and structured in a way to contain a data space purposed for the authentication data. Before processing the packet with the HMAC computation the data space is filled with some data a receiver of the packet can reproduce exactly, typically involving an arbitrary number known as a padding constant. After the HMAC computation the data space inside the packet is overwritten with the resulting authentication data.

The padding constant used in [RIP2-AUTH] is 0x878FE1F3 four-octet value. Subsequent works (including [OSPF2-AUTH] and [OSPF3-AUTH]) inherited both the method and the padding constant value. In particular, [OSPF3-AUTH] uses a source IPv6 address to set the first 16 octets of the padded area and the padding constant to set any subsequent octets. This mechanism uses the source IPv6 address in the same way, but the padding constant size and value are different.

Since any fixed arbitrary value of a padding constant does not affect cryptographic characteristics of a hash algorithm and the HMAC construct, and since single-octet padding is more straightforward to implement, the padding constant used by this mechanism is 0x00 single-octet value. This is respectively addressed in sending (Section 5.3 item 5) and receiving (Section 5.4 item 6) procedures.

2.3. Cryptographic Sequence Number Specifics

Operation of this mechanism may involve multiple local and multiple remote cryptographic sequence numbers, each essentially being a 48-bit unsigned integer. This specification uses a term "TS/PC number" to avoid confusion with the route's sequence number of the original Babel specification (Section 2.5 of [BABEL]) and to stress the fact that there are two distinguished parts of this 48-bit number, each handled in its specific way (see Section 5.1):



The high-order 32 bits are called "timestamp" (TS) and the low-order 16 bits are called "packet counter" (PC).

This mechanism stores, updates, compares, and encodes each TS/PC

number as two independent unsigned integers, TS and PC respectively. Such comparison of TS/PC numbers performed in item 3 of Section 5.4 is algebraically equivalent to comparison of respective 48-bit unsigned integers. Any byte order conversion, when required, is performed on TS and PC parts independently.

2.4. Definition of HMAC

The algorithm description below uses the following nomenclature, which is consistent with [FIPS-198]:

Text Is the data on which the HMAC is calculated (note item (b) of Section 9). In this specification it is the contents of a Babel packet ranging from the beginning of the Magic field of the Babel packet header to the end of the last octet of the Packet Body field, as defined in Section 4.2 of [BABEL] (see Figure 2).

H Is the specific hash algorithm (see Section 2.1).

K Is a sequence of octets of an arbitrary, known length.

Ko Is the cryptographic key used with the hash algorithm.

B Is the block size of H, measured in octets rather than bits. Note that B is the internal block size, not the digest length.

L Is the digest length of H, measured in octets rather than bits.

XOR Is the bitwise exclusive-or operation.

Opad Is the hexadecimal value 0x5c repeated B times.

Ipad Is the hexadecimal value 0x36 repeated B times.

The algorithm below is the original, unmodified HMAC construct as defined in both [RFC2104] and [FIPS-198], hence it is different from the algorithms defined in [RIP2-AUTH], [OSPF2-AUTH], and [OSPF3-AUTH] in exactly two regards:

- The algorithm below sets the size of Ko to B, not to L (L is not greater than B). This resolves both ambiguity in XOR expressions and incompatibility in handling of keys having length greater than L but not greater than B.

- The algorithm below does not change value of Text before or after the computation. Both padding of a Babel packet before the

computation and placing of the result inside the packet are performed elsewhere.

The intent of this is to enable the most straightforward use of cryptographic libraries by implementations of this specification. At the time of this writing implementations of the original HMAC construct coupled with hash algorithms of choice are generally available.

Description of the algorithm:

1. Preparation of the Key

   In this application, Ko is always B octets long. If K is B octets long, then Ko is set to K. If K is more than B octets long, then Ko is set to H(K) with the necessary amount of zeroes appended to the end of H(K), such that Ko is B octets long. If K is less than B octets long, then Ko is set to K with zeroes appended to the end of K, such that Ko is B octets long.

2. First-Hash

   A First-Hash, also known as the inner hash, is computed as follows:

   First-Hash = H(Ko XOR Ipad || Text)

3. Second-Hash

   A second hash, also known as the outer hash, is computed as follows:

   Second-Hash = H(Ko XOR Opad || First-Hash)

4. Result

   The resulting Second-Hash becomes the authentication data that is returned as the result of HMAC calculation.

3. Updates to Protocol Data Structures

3.1. RxAuthRequired

RxAuthRequired is a boolean parameter, its default value MUST be TRUE. An implementation SHOULD make RxAuthRequired a per-interface parameter, but MAY make it specific to the whole protocol instance. The conceptual purpose of RxAuthRequired is to enable a smooth

migration from an unauthenticated to an authenticated Babel packet exchange and back (see Section 7.3). Current value of RxAuthRequired directly affects the receiving procedure defined in Section 5.4. An implementation SHOULD allow the operator to change RxAuthRequired value at runtime or by means of Babel speaker restart. An implementation MUST allow the operator to discover the effective value of RxAuthRequired at runtime or from the system documentation.

3.2. LocalTS

LocalTS is a 32-bit unsigned integer variable, it is the TS part of a per-interface TS/PC number. LocalTS is a strictly per-interface variable not intended to be changed by the operator. Its initialization is explained in Section 5.1.

3.3. LocalPC

LocalPC is a 16-bit unsigned integer variable, it is the PC part of a per-interface TS/PC number. LocalPC is a strictly per-interface variable not intended to be changed by the operator. Its initialization is explained in Section 5.1.

3.4. MaxDigestsIn

MaxDigestsIn is an unsigned integer parameter conceptually purposed for limiting the amount of CPU time spent processing a received authenticated packet. The receiving procedure performs the most CPU-intensive operation, the HMAC computation, only at most MaxDigestsIn (Section 5.4 item 7) times for a given packet.

MaxDigestsIn value MUST be at least 2. An implementation SHOULD make MaxDigestsIn a per-interface parameter, but MAY make it specific to the whole protocol instance. An implementation SHOULD allow the operator to change the value of MaxDigestsIn at runtime or by means of Babel speaker restart. An implementation MUST allow the operator to discover the effective value of MaxDigestsIn at runtime or from the system documentation.

3.5. MaxDigestsOut

MaxDigestsOut is an unsigned integer parameter conceptually purposed for limiting the amount of a sent authenticated packet’s space spent on authentication data. The sending procedure adds at most MaxDigestsOut (Section 5.3 item 5) HMAC results to a given packet, concurring with the output buffer management explained in Section 6.2.

The MaxDigestsOut value MUST be at least 2. An implementation SHOULD

make MaxDigestsOut a per-interface parameter, but MAY make it specific to the whole protocol instance. An implementation SHOULD allow the operator to change the value of MaxDigestsOut at runtime or by means of Babel speaker restart, in a safe range. The maximum safe value of MaxDigestsOut is implementation-specific (see Section 6.2). An implementation MUST allow the operator to discover the effective value of MaxDigestsOut at runtime or from the system documentation.

3.6. ANM Table

The ANM (Authentic Neighbours Memory) table resembles the neighbour table defined in Section 3.2.3 of [BABEL]. Note that the term "neighbour table" means the neighbour table of the original Babel specification, and the term "ANM table" means the table defined herein. Indexing of the ANM table is done in exactly the same way as indexing of the neighbour table, but purpose, field set and associated procedures are different.

The conceptual purpose of the ANM table is to provide longer term replay attack protection than it would be possible using the neighbour table. Expiry of an inactive entry in the neighbour table depends on the last received Hello Interval of the neighbour and typically stands for tens to hundreds of seconds (see Appendix A and Appendix B of [BABEL]). Expiry of an inactive entry in the ANM table depends only on the local speaker’s configuration. The ANM table retains (for at least the amount of seconds set by ANM timeout parameter defined in Section 3.7) a copy of TS/PC number advertised in authentic packets by each remote Babel speaker.

The ANM table is indexed by pairs of the form (Interface, Source). Every table entry consists of the following fields:

- Interface

  An implementation-specific reference to the local node’s interface that the authentic packet was received through.

- Source

  IPv6 source address of the Babel speaker that the authentic packet was received from.

- LastTS

  A 32-bit unsigned integer, the TS part of a remote TS/PC number.

- LastPC

A 16-bit unsigned integer, the PC part of a remote TS/PC number.

Each ANM table entry has an associated aging timer, which is reset by the receiving procedure (Section 5.4 item 9). If the timer expires, the entry is deleted from the ANM table.

An implementation SHOULD use a persistent memory (NVRAM) to retain the contents of ANM table across restarts of the Babel speaker, but only as long as both the Interface field reference and expiry of the aging timer remain correct. An implementation MUST make it clear, if and how persistent memory is used for ANM table. An implementation SHOULD allow the operator to retrieve the current contents of ANM table at runtime. An implementation SHOULD allow the operator to remove some or all of ANM table entries at runtime or by means of Babel speaker restart.

3.7. ANM Timeout

ANM timeout is an unsigned integer parameter. An implementation SHOULD make ANM timeout a per-interface parameter, but MAY make it specific to the whole protocol instance. ANM timeout is conceptually purposed for limiting the maximum age (in seconds) of entries in the ANM table standing for inactive Babel speakers. The maximum age is immediately related to replay attack protection strength. The strongest protection is achieved with the maximum possible value of ANM timeout set, but it may not provide the best overall result for specific network segments and implementations of this mechanism.

In the first turn, implementations unable to maintain local TS/PC number strictly increasing across Babel speaker restarts will reuse the advertised TS/PC numbers after each restart (see Section 5.1). The neighbouring speakers will treat the new packets as replayed and discard them until the aging timer of respective ANM table entry expires or the new TS/PC number exceeds the one stored in the entry.

Another possible, but less probable, case could be an environment involving physical moves of network interfaces hardware between routers. Even performed without restarting Babel speakers, these would cause random drops of the TS/PC number advertised for a given (Interface, Source) index, as viewed by neighbouring speakers, since IPv6 link-local addresses are typically derived from interface hardware addresses.

Assuming that in such cases the operators would prefer to use a lower ANM timeout value to let the entries expire on their own rather than having to manually remove them from the ANM table each time, an implementation SHOULD set the default value of ANM timeout to a value between 30 and 300 seconds.

At the same time, network segments may exist with every Babel speaker having its advertised TS/PC number strictly increasing over the deployed lifetime. Assuming that in such cases the operators would prefer using a much higher ANM timeout value, an implementation SHOULD allow the operator to change the value of ANM timeout at runtime or by means of Babel speaker restart. An implementation MUST allow the operator to discover the effective value of ANM timeout at runtime or from the system documentation.

3.8. Configured Security Associations

A Configured Security Association (CSA) is a data structure conceptually purposed for associating authentication keys and hash algorithms with Babel interfaces. All CSAs are managed in finite sequences, one sequence per interface ("interface's sequence of CSAs" hereafter). Each interface's sequence of CSAs, as an integral part of the Babel speaker configuration, MAY be intended for a persistent storage as long as this conforms with the implementation's key management policy. The default state of an interface's sequence of CSAs is empty, which has a special meaning of no authentication configured for the interface. The sending (Section 5.3 item 1) and the receiving (Section 5.4 item 1) procedures address this convention accordingly.

A single CSA structure consists of the following fields:

- HashAlgo

  An implementation-specific reference to one of the hash algorithms supported by this implementation (see Section 2.1).

- KeyChain

  A finite sequence of elements ("KeyChain sequence" hereafter) representing authentication keys, each element being a structure consisting of the following fields:

  * LocalKeyID

    An unsigned integer of an implementation-specific bit length.

  * AuthKeyOctets

    A sequence of octets of an arbitrary, known length to be used as the authentication key.

  * KeyStartAccept

The time that this Babel speaker will begin considering this authentication key for accepting packets with authentication data.

* KeyStartGenerate

  The time that this Babel speaker will begin considering this authentication key for generating packet authentication data.

* KeyStopGenerate

  The time that this Babel speaker will stop considering this authentication key for generating packet authentication data.

* KeyStopAccept

  The time that this Babel speaker will stop considering this authentication key for accepting packets with authentication data.

Since there is no limit imposed on the number of CSAs per interface, but the number of HMAC computations per sent/received packet is limited (through MaxDigestsOut and MaxDigestsIn respectively), only a fraction of the associated keys and hash algorithms may appear used in the process. The ordering of elements within a sequence of CSAs and within a KeyChain sequence is important to make the association selection process deterministic and transparent. Once this ordering is deterministic at the Babel interface level, the intermediate data derived by the procedure defined in Section 5.2 will be deterministically ordered as well.

An implementation SHOULD allow an operator to set any arbitrary order of elements within a given interface’s sequence of CSAs and within the KeyChain sequence of a given CSA. Regardless if this requirement is or isn’t met, the implementation MUST provide a mean to discover the actual element order used. Whichever order is used by an implementation, it MUST be preserved across Babel speaker restarts.

Note that none of the CSA structure fields is constrained to contain unique values. Section 6.4 explains this in more detail. It is possible for the KeyChain sequence to be empty, although this is not the intended manner of CSAs use.

The KeyChain sequence has a direct prototype, which is the "key chain" syntax item of some existing router configuration languages. Whereas an implementation already implements this syntax item, it is suggested to reuse it, that is, to implement a CSA syntax item referring to a key chain item instead of reimplementing the latter in

full.

3.9. Effective Security Associations

An Effective Security Association (ESA) is a data structure immediately used in sending (Section 5.3) and receiving (Section 5.4) procedures. Its conceptual purpose is to determine a runtime interface between those procedures and the deriving procedure defined in Section 5.2. All ESAs are temporary data units managed as elements of finite sequences that are not intended for a persistent storage. Element ordering within each such finite sequence ("sequence of ESAs" hereafter) MUST be preserved as long as the sequence exists.

A single ESA structure consists of the following fields:

- HashAlgo

  An implementation-specific reference to one of the hash algorithms supported by this implementation (see Section 2.1).

- KeyID

  A 16-bit unsigned integer.

- AuthKeyOctets

  A sequence of octets of an arbitrary, known length to be used as the authentication key.

Note that among the protocol data structures introduced by this mechanism ESA is the only one not directly interfaced with the system operator (see Figure 1), it is not immediately present in the protocol encoding either. However, ESA is not just a possible implementation technique, but an integral part of this specification: the deriving (Section 5.2), the sending (Section 5.3), and the receiving (Section 5.4) procedures are defined in terms of the ESA structure and its semantics provided herein. ESA is as meaningful for a correct implementation as the other protocol data structures.

4. Updates to Protocol Encoding

4.1. Justification

Choice of encoding is very important in the long term. The protocol encoding limits various authentication mechanism designs and encodings, which in turn limit future developments of the protocol.

Considering existing implementations of Babel protocol instance itself and related modules of packet analysers, the current encoding of Babel allows for compact and robust decoders. At the same time, this encoding allows for future extensions of Babel by three (not excluding each other) principal means defined by Section 4.2 and Section 4.3 of [BABEL]:

a. A Babel packet consists of a four-octet header followed by a packet body, that is, a sequence of TLVs (see Figure 2). Besides the header and the body, an actual Babel datagram may have an arbitrary amount of trailing data between the end of the packet body and the end of the datagram. An instance of the original protocol silently ignores such trailing data.

b. The packet body employs a binary format allowing for 256 TLV types and imposing no requirements on TLV ordering or number of TLVs of a given type in a packet. Only TLV length matters within the packet body, TLV body contents is to be interpreted elsewhere. This makes an iteration over the sequence of TLVs possible without knowledge of the body structure of each TLV (with the only distinction between a Pad1 TLV and any other TLVs). The original specification allocates TLV types 0 through 10 (see Table 1) and defines TLV body structure for each. An instance of the original protocol silently ignores any unknown TLV types.

c. Within each TLV of the packet body there may be some "extra data" after the "expected length" of the TLV body. An instance of the original protocol silently ignores any such extra data. Note that any TLV types without the expected length defined (such as PadN TLV) cannot be extended with the extra data.

Considering each principal extension mean for the specific purpose of adding authentication data items to each protocol packet, the following arguments can be made:

- Use of the TLV extra data of some existing TLV type would not be a solution, since no particular TLV type is guaranteed to be present in a Babel packet.

- Use of the TLV extra data could also conflict with future developments of the protocol encoding.

- Since the packet trailing data is currently unstructured, using it would involve defining an encoding structure and associated procedures, adding to the complexity of both specification and implementation and increasing the exposure to protocol attacks such as fuzzing.

- A naive use of the packet trailing data would make it unavailable to any future extension of Babel. Since this mechanism is possibly not the last extension and since some other extensions may allow no other embedding means except the packet trailing data, the defined encoding structure would have to enable multiplexing of data items belonging to different extensions. Such a definition is out of the scope of this work.

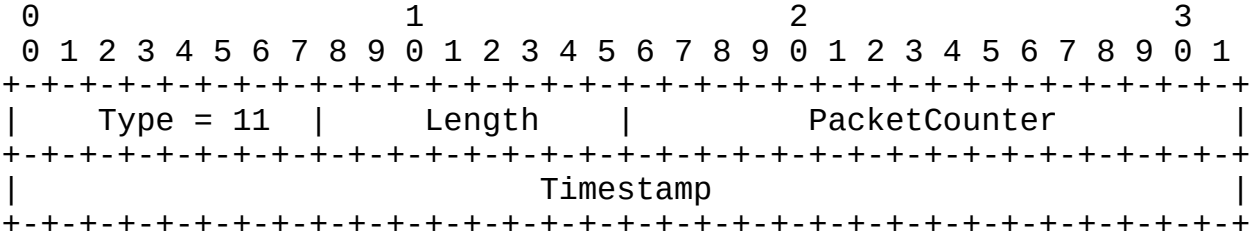
- Deprecating an extension (or only its protocol encoding) that uses purely purpose-allocated TLVs is as simple as deprecating the TLVs.

- Use of purpose-allocated TLVs is transparent for both the original protocol and any its future extensions, regardless of the embedding mean(s) used by the latter.

Considering all of the above, this mechanism neither uses the packet trailing data nor uses the TLV extra data, but uses two new TLV types: type 11 for a TS/PC number and type 12 for an HMAC result (see Table 1).

4.2. TS/PC TLV

The purpose of a TS/PC TLV is to store a single TS/PC number. There is normally exactly one TS/PC TLV in an authenticated Babel packet. Any occurences of this TLV except the first are ignored.



Fields:

Type Set to 11 to indicate a TS/PC TLV.

Length The length of the body, exclusive of the Type and Length fields.

PacketCounter A 16-bit unsigned integer in network byte order, the PC part of a TS/PC number stored in this TLV.

Timestamp A 32-bit unsigned integer in network byte order, the TS part of a TS/PC number stored in this TLV.

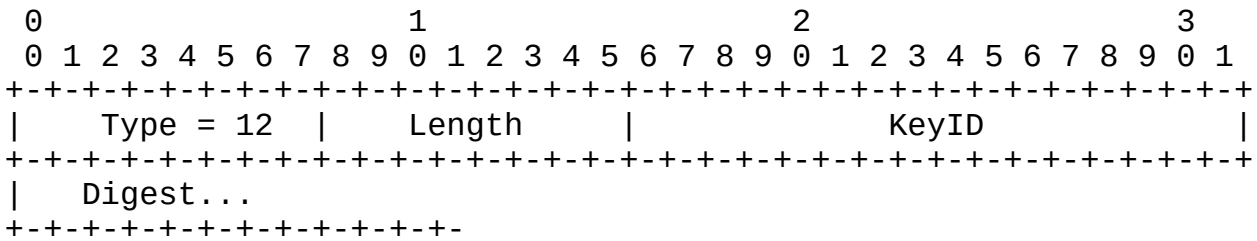
Note that the ordering of PacketCounter and Timestamp in the TLV structure is opposite to the ordering of TS and PC in "TS/PC" term and the 48-bit equivalent (see Section 2.3).

Considering the "expected length" and the "extra data" in the definition of Section 4.2 of [BABEL], the expected length of a TS/PC TLV body is unambiguously defined as 6 octets. The receiving procedure correctly processes any TS/PC TLV with body length not less than the expected, ignoring any extra data (Section 5.4 items 3 and 9). The sending procedure produces a TS/PC TLV with body length equal to the expected and Length field set respectively (Section 5.3 item 3).

Future Babel extensions (such as sub-TLVs) MAY modify the sending procedure to include the extra data after the fixed-size TS/PC TLV body defined herein, making necessary adjustments to Length TLV field, "Body length" packet header field and output buffer management explained in Section 6.2.

4.3. HMAC TLV

The purpose of an HMAC TLV is to store a single HMAC result. To assist a receiver in reproducing the HMAC computation, LocalKeyID modulo 2^16 of the authentication key is also provided in the TLV. There is normally at least one HMAC TLV in an authenticated Babel packet.



Fields:

Type Set to 12 to indicate an HMAC TLV.

Length The length of the body, exclusive of the Type and Length fields.

KeyID A 16-bit unsigned integer in network byte order.

Digest A variable-length sequence of octets.

The Digest field of the TLV MUST be at least 16 octets long to make sure there is enough space to be padded with a full IPv6 address (see Section 5.3 item 5, Section 5.4 item 6, Table 3 and Table 4). At the time of this writing an instance of the Babel protocol uses only IPv6 link-local addresses as the source address of the protocol packets (see Section 3.1 of [BABEL]) and the first 8 octets of the address always stand for the fe80::/64 prefix. However, if a future protocol extension needs to send Babel packets from a source address outside of the fe80::/64 prefix, as little changes to this mechanism as possible should be required to authenticate such packets as well. For this reason all 16 octets of the IPv6 address are used for the padding.

Considering the "expected length" and the "extra data" in the definition of Section 4.2 of [BABEL], the expected length of an HMAC TLV body is not defined. The receiving procedure processes every octet of the Digest field, deriving the field boundary from the Length field value (Section 5.4 item 6). The sending procedure produces HMAC TLVs with Length field precisely sizing the Digest field to match digest length of the hash algorithm used (Section 5.3 items 5 and 8).

The HMAC TLV structure defined herein is final, future Babel extensions MUST NOT extend it with any extra data.

5. Updates to Protocol Operation

5.1. Per-Interface TS/PC Number Updates

The LocalTS and LocalPC interface-specific variables constitute the TS/PC number of a Babel interface. This number is advertised in the TS/PC TLV of authenticated Babel packets sent from that interface. There is only one property mandatory for the advertised TS/PC number: its 48-bit equivalent (see Section 2.3) MUST be strictly increasing within the scope of a given interface of a Babel speaker as long as the protocol instance is continuously operating. This property combined with ANM tables of neighbouring Babel speakers provides them with the most basic replay attack protection.

Initialization and increment are two principal updates performed on an interface TS/PC number. The initialization is performed when a new interface becomes a part of a Babel protocol instance. The increment is performed by the sending procedure (Section 5.3 item 2)

before advertising the TS/PC number in a TS/PC TLV.

Depending on particular implementation method of these two updates the advertised TS/PC number may possess additional properties improving the replay attack protection strength. This includes, but is not limited to the methods below.

a. The most straightforward implementation would use LocalTS as a plain wrap counter, defining the updates as follows:

   initialization Set LocalPC to 0, set LocalTS to 0.

   increment Increment LocalPC by 1. If LocalPC wraps (0xFFFF + 1 = 0x0000), increment LocalTS by 1.

   In this case the advertised TS/PC numbers would be reused after each Babel protocol instance restart, making neighbouring speakers reject authenticated packets until the respective ANM table entries expire or the new TS/PC number exceeds the old (see Section 3.6 and Section 3.7).

b. A more advanced implementation could make a use of any 32-bit unsigned integer timestamp (number of time units since an arbitrary epoch) such as the UNIX timestamp, whereas the timestamp itself spans a reasonable time range and is guaranteed against a decrease (such as one resulting from network time use). The updates would be defined as follows:

   initialization Set LocalPC to 0, set LocalTS to 0.

   increment If the current timestamp is greater than LocalTS, set LocalTS to the current timestamp and LocalPC to 0, then consider the update complete. Otherwise increment LocalPC by 1 and, if LocalPC wraps, increment LocalTS by 1.

   In this case the advertised TS/PC number would remain unique across the speaker’s deployed lifetime without the need for any persistent storage. However, a suitable timestamp source is not available in every implementation case.

c. Another advanced implementation could use LocalTS in a way similar to the "wrap/boot counter" suggested in Section 4.1.1 of [OSPF3-AUTH], defining the updates as follows:

initialization Set LocalPC to 0. Whether there is a TS value stored in NVRAM for the current interface, set LocalTS to the stored TS value, then increment the stored TS value by 1. Otherwise set LocalTS to 0 and set the stored TS value to 1.

increment Increment LocalPC by 1. If LocalPC wraps, set LocalTS to the TS value stored in NVRAM for the current interface, then increment the stored TS value by 1.

In this case the advertised TS/PC number would also remain unique across the speaker's deployed lifetime, relying on NVRAM for storing multiple TS numbers, one per interface.

As long as the TS/PC number retains its mandatory property stated above, it is up to the implementor, which TS/PC number updates methods are available and if the operator can configure the method per-interface and/or at runtime. However, an implementation MUST disclose the essence of each update method it includes, in a comprehensible form such as natural language description, pseudocode, or source code. An implementation MUST allow the operator to discover, which update method is effective for any given interface, either at runtime or from the system documentation. These requirements are necessary to enable the optimal (see Section 3.7) management of ANM timeout in a network segment.

Note that wrapping (0xFFFFFFFF + 1 = 0x00000000) of LastTS is unlikely, but possible, causing the advertised TS/PC number to be reused. Resolving this situation requires replacing all authentication keys of the involved interface. In addition to that, if the wrap was caused by a timestamp reaching its end of epoch, using this mechanism will be impossible for the involved interface until some different timestamp or update implementation method is used.

5.2. Deriving ESAs from CSAs

Neither receiving nor sending procedures work with the contents of interface's sequence of CSAs directly, both (Section 5.4 item 4 and Section 5.3 item 4 respectively) derive a sequence of ESAs from the sequence of CSAs and use the derived sequence (see Figure 1). There are two main goals achieved through this indirection:

- Elimination of expired authentication keys and deduplication of security associations. This is done as early as possible to keep subsequent procedures focused on their respective tasks.

- Maintenance of particular ordering within the derived sequence of ESAs. The ordering deterministically depends on the ordering within the interface's sequence of CSAs and the ordering within KeyChain sequence of each CSA. The particular correlation maintained by this procedure implements a concept of fair (independent of number of keys contained by each) competition between CSAs.

The deriving procedure uses the following input arguments:

- input sequence of CSAs
- direction (sending or receiving)
- current time (CT)

The processing of input arguments begins with an empty output sequence of ESAs and consists of the following steps:

1. Make a temporary copy of the input sequence of CSAs.
2. Remove all expired authentication keys from each KeyChain sequence of the copy, that is, any keys such that:
    * for receiving: KeyStartAccept is greater than CT or KeyStopAccept is less than CT
    * for sending: KeyStartGenerate is greater than CT or KeyStopGenerate is less than CT

    Note well that there are no special exceptions. Remove all expired keys, even if there are no keys left after that (see Section 7.4).
3. Use the copy to populate the output sequence of ESAs as follows:
    1. Whether the KeyChain sequence of the first CSA contains at least one key, use its first key to produce an ESA with fields set as follows:

        HashAlgo — Set to HashAlgo of the current CSA.

        KeyID — Set to LocalKeyID modulo 2^16 of the current key of the current CSA.

AuthKeyOctets Set to AuthKeyOctets of the current key of the current CSA.

Append this ESA to the end of the output sequence.

2. When the KeyChain sequence of the second CSA contains at least one key, use its first key the same way and so forth until all first keys of the copy are processed.

3. When the KeyChain sequence of the first CSA contains at least two keys, use its second key the same way.

4. When the KeyChain sequence of the second CSA contains at least two keys, use its second key the same way and so forth until all second keys of the copy are processed.

5. And so forth until all keys of all CSAs of the copy are processed, exactly once each.

In the description above the ordinals ("first", "second", and so on) with regard to keys stand for an element position after the removal of expired keys, not before. For example, if a KeyChain sequence was { Ka, Kb, Kc, Kd } before the removal and became { Ka, Kd } after, then Ka would be the "first" element and Kd would be the "second".

4. Deduplicate the ESAs in the output sequence, that is, wherever two or more ESAs exist that share the same (HashAlgo, KeyID, AuthKeyOctets) triplet value, remove all of these ESAs except the one closest to the beginning of the sequence.

The resulting sequence will contain zero or more unique ESAs, ordered in a way deterministically correlated with ordering of CSAs within the original input sequence of CSAs and ordering of keys within each KeyChain sequence. This ordering maximizes the probability of having equal amount of keys per original CSA in any N first elements of the resulting sequence. Possible optimisations of this deriving procedure are outlined in Section 6.3.

5.3. Updates to Packet Sending

Perform the following authentication-specific processing after the instance of the original protocol considers an outgoing Babel packet ready for sending, but before the packet is actually sent (see Figure 1). After that send the packet regardless if the authentication-specific processing modified the outgoing packet or left it intact.

1. If the current outgoing interface's sequence of CSAs is empty, finish authentication-specific processing and consider the packet ready for sending.

2. Increment TS/PC number of the current outgoing interface as explained in Section 5.1.

3. Append to the end of the packet body (see Figure 2) a TS/PC TLV with fields set as follows:

   Type — Set to 11.

   Length — Set to 6.

   PacketCounter — Set to the current value of LocalPC variable of the current outgoing interface.

   Timestamp — Set to the current value of LocalTS variable of the current outgoing interface.

   Note that the current step may involve byte order conversion.

4. Derive a sequence of ESAs using procedure defined in Section 5.2 with the current interface's sequence of CSAs as the input sequence of CSAs, the current time as CT and "sending" as the direction. Proceed to the next step even if the derived sequence is empty.

5. Iterate over the derived sequence using its ordering. For each ESA append to the end of the packet body (see Figure 2) an HMAC TLV with fields set as follows:

   Type — Set to 12.

   Length — Set to 2 plus digest length of HashAlgo of the current ESA.

   KeyID — Set to KeyID of the current ESA.

   Digest — Size exactly equal to the digest length of HashAlgo of the current ESA. Set the first 16 octets to the source IPv6 address of the current packet (see Section 6.1) and any subsequent octets to 0x00 (see Table 3).

   As soon as there are MaxDigestsOut HMAC TLVs appended to the current packet body, immediately proceed to the next step.

   Note that the current step may involve byte order conversion.

6. Increment the "Body length" field value of the current packet header by the total length of TS/PC and HMAC TLVs appended to the current packet body so far.

   Note that the current step may involve byte order conversion.

7. Make a temporary copy of the current packet.

8. Iterate over the derived sequence again, using the same order and number of elements. For each ESA (and respectively for each HMAC TLV recently appended to the current packet body) compute an HMAC result (see Section 2.4) using the temporary copy (not the original packet) as Text, HashAlgo of the current ESA as H, and AuthKeyOctets of the current ESA as K. Write the HMAC result to the Digest field of the current HMAC TLV (see Table 4) of the current packet (not the copy).

9. After this point, allow no more changes to the current packet header and body and consider it ready for sending.

Note that even when the derived sequence of ESAs is empty, the packet is sent anyway with only a TS/PC TLV appended to its body. Although such a packet would not be authenticated, the presence of the sole TS/PC TLV would indicate authentication key exhaustion to operators of neighbouring Babel speakers. See also Section 7.4.

5.4. Updates to Packet Receiving

Perform the following authentication-specific processing after an incoming Babel packet is received from the local IPv6 stack, but before it is processed by the Babel protocol instance (see Figure 1). The final action conceptually depends not only upon the result of the authentication-specific processing, but also on the current value of RxAuthRequired parameter. Immediately after any processing step below accepts or refuses the packet, either deliver the packet to the instance of the original protocol (when the packet is accepted or RxAuthRequired is FALSE) or discard it (when the packet is refused and RxAuthRequired is TRUE).

1. If the current incoming interface’s sequence of CSAs is empty, accept the packet.

2. If the current packet does not contain a TS/PC TLV, refuse it.

3. Perform a lookup in the ANM table for an entry having Interface equal to the current incoming interface and Source equal to the source address of the current packet. If such an entry does not exist, immediately proceed to the next step. Otherwise, compare

the entry's LastTS and LastPC field values with Timestamp and PacketCounter values respectively of the first TS/PC TLV of the packet. That is, refuse the packet, if at least one of the following two conditions is true:

* Timestamp is less than LastTS

* Timestamp is equal to LastTS and PacketCounter is not greater than LastPC

Note that the current step may involve byte order conversion.

4. Derive a sequence of ESAs using procedure defined in Section 5.2 with the current interface's sequence of CSAs as the input sequence of CSAs, current time as CT and "receiving" as the direction. If the derived sequence is empty, refuse the packet.

5. Make a temporary copy of the current packet.

6. For every HMAC TLV present in the temporary copy (not the original packet) pad all octets of its Digest field using the source IPv6 address of the current packet to set the first 16 octets and 0x00 to set any subsequent octets (see Table 3).

7. Iterate over all the HMAC TLVs of the original input packet (not the copy) using their order of appearance in the packet. For each HMAC TLV look up all ESAs in the derived sequence such that 2 plus digest length of HashAlgo of the ESA is equal to Length of the TLV and KeyID of the ESA is equal to value of KeyID of the TLV. Iterate over these ESAs in the relative order of their appearance on the full sequence of ESAs. Note that nesting the iterations the opposite way (over ESAs, then over HMAC TLVs) would be wrong.

   For each of these ESAs compute an HMAC result (see Section 2.4) using the temporary copy (not the original packet) as Text, HashAlgo of the current ESA as H, and AuthKeyOctets of the current ESA as K. If the current HMAC result exactly matches the contents of Digest field of the current HMAC TLV, immediately proceed to the next step. Otherwise, if the number of HMAC computations done for the current packet so far is equal to MaxDigestsIn, immediately proceed to the next step. Otherwise follow the normal order of iterations.

   Note that the current step may involve byte order conversion.

8. Refuse the input packet unless there was a matching HMAC result in the previous step.

9. Modify the ANM table, using the same index as for the entry lookup above, to contain an entry with LastTS set to the value of Timestamp and LastPC set to the value of PacketCounter fields of the first TS/PC TLV of the current packet. That is, either add a new ANM table entry or update the existing one, depending on the result of the entry lookup above. Reset the entry's aging timer to the current value of ANM timeout.

   Note that the current step may involve byte order conversion.

10. Accept the input packet.

Note that RxAuthRequired affects only the final action, but not the defined flow of authentication-specific processing. The purpose of this is to preserve authentication-specific processing feedback (such as log messages and event counters updates) even with RxAuthRequired set to FALSE. This allows an operator to predict the effect of changing RxAuthRequired from FALSE to TRUE during a migration scenario (Section 7.3) implementation.

5.5. Authentication-Specific Statistics Maintenance

A Babel speaker implementing this mechanism SHOULD maintain a set of counters for the following events, per protocol instance and per interface:

- Sending of an unauthenticated Babel packet through an interface having an empty sequence of CSAs (Section 5.3 item 1).
- Sending of an unauthenticated Babel packet with a TS/PC TLV but without any HMAC TLVs due to an empty derived sequence of ESAs (Section 5.3 item 4).
- Sending of an authenticated Babel packet containing both TS/PC and HMAC TLVs (Section 5.3 item 9).
- Accepting of a Babel packet received through an interface having an empty sequence of CSAs (Section 5.4 item 1).
- Refusing of a received Babel packet due to an empty derived sequence of ESAs (Section 5.4 item 4).
- Refusing of a received Babel packet missing any TS/PC TLVs (Section 5.4 item 2).
- Refusing of a received Babel packet due to the first TS/PC TLV failing the ANM table check (Section 5.4 item 3).

- Refusing of a received Babel packet missing any HMAC TLVs (Section 5.4 item 8).
- Refusing of a received Babel packet due to none of the processed HMAC TLVs passing the ESA check (Section 5.4 item 8).
- Accepting of a received Babel packet having both TS/PC and HMAC TLVs (Section 5.4 item 10).
- Delivery of a refused packet to the instance of the original protocol due to RxAuthRequired parameter set to FALSE.

Note that terms "accepting" and "refusing" are used in the sense of the receiving procedure, that is, "accepting" does not mean a packet delivered to the instance of the original protocol purely because the RxAuthRequired parameter is set to FALSE. Event counters readings SHOULD be available to the operator at runtime.

6. Implementation Notes

6.1. IPv6 Source Address Selection for Sending

Section 3.1 of [BABEL] defines that Babel datagrams are exchanged using IPv6 link-local address as source address. This implies having at least one such address assigned to an interface participating in the exchange. When the interface has more than one link-local addresses assigned, selection of one particular link-local address as packet source address is left up to the local IPv6 stack, since this choice is not meaningful in the scope of the original protocol. However, the sending procedure requires exact knowledge of packet source address for proper padding (Section 5.3 item 5) of HMAC TLVs.

As long as a Babel interface has more than one IPv6 link-local addresses assigned, the Babel speaker SHOULD internally choose one particular link-local address for Babel packet sending purposes and make this choice to both the sending procedure and local IPv6 stack (see Figure 1). Wherever this requirement cannot be met, this limitation MUST be clearly stated in the system documentation to allow an operator to plan IPv6 address management accordingly.

6.2. Output Buffer Management

An instance of the original protocol buffers produced TLVs until the buffer becomes full or a delay timer has expired or an urgent TLV is produced. This is performed independently for each Babel interface with each buffer sized according to the interface MTU (see Sections 3.1 and 4 of [BABEL]).

Since TS/PC and HMAC TLVs and any other TLVs, in the first place those of the original protocol, share the same packet space (see Figure 2) and respectively the same buffer space, a particular portion of each interface buffer needs to be reserved for 1 TS/PC TLV and up to MaxDigestsOut HMAC TLVs. The amount (R) of this reserved buffer space is calculated as follows:

```
R = St + MaxDigestsOut * Sh =
  = 8  + MaxDigestsOut * (4 + Lmax)
```

St Is the size of a TS/PC TLV.

Sh Is the size of an HMAC TLV.

Lmax Is the maximum digest length in octets possible for a particular interface. It SHOULD be calculated based on particular interface's sequence of CSAs, but MAY be taken as the maximum digest length supported by particular implementation.

An implementation allowing for per-interface value of MaxDigestsOut parameter has to account for different value of R across different interfaces, even having the same MTU. An implementation allowing for runtime change of MaxDigestsOut parameter value has to take care of the TLVs already buffered by the time of the change, especially when the change increases the value of R.

The maximum safe value of MaxDigestsOut parameter depends on the interface MTU and maximum digest length used. In general, at least 200-300 octets of a Babel packet should be always available to data other than TS/PC and HMAC TLVs. An implementation following the requirements of Section 4 of [BABEL] would send packets sized 512 octets or larger. If, for example, the maximum digest length is 64 octets and MaxDigestsOut value is 4, the value of R would be 280, leaving less than a half of a 512-octet packet for any other TLVs. As long as the interface MTU is larger or digest length is smaller, higher values of MaxDigestsOut can be used safely.

6.3. Optimisations of ESAs Deriving

The following optimisations of the ESAs deriving procedure can reduce amount of CPU time consumed by authentication-specific processing, preserving an implementation's effective behaviour.

a. The most straightforward implementation would treat the deriving procedure as a per-packet action. But since the procedure is deterministic (its output depends on its input only), it is possible to significantly reduce the number of times the

procedure is performed.

The procedure would obviously return the same result for the same input arguments (sequence of CSAs, direction, CT) values. However, it is possible to predict when the result will remain the same even for a different input. That is, when the input sequence of CSAs and the direction both remain the same but CT changes, the result will remain the same as long as CT's order on the time axis (relative to all critical points of the sequence of CSAs) remains unchanged. Here, the critical points are KeyStartAccept and KeyStopAccept (for the "receiving" direction) and KeyStartGenerate and KeyStopGenerate (for the "sending" direction) of all keys of all CSAs of the input sequence. In other words, in this case the result will remain the same as long as both none of the active keys expire and none of the inactive keys enter into operation.

An implementation optimised this way would perform the full deriving procedure for a given (interface, direction) pair only after an operator's change to the interface's sequence of CSAs or after reaching one of the critical points mentioned above.

b. Considering that the sending procedure iterates over at most MaxDigestsOut elements of the derived sequence of ESAs (Section 5.3 item 5), there would be little sense in the case of "sending" direction in returning more than MaxDigestsOut unique ESAs in the derived sequence. Note that a similar optimisation is impossible in the case of "receiving" direction, since number of ESAs actually used in examining a particular packet cannot be determined in advance.

6.4. Security Associations Duplication

This specification defines three data structures as finite sequences: a KeyChain sequence, an interface's sequence of CSAs, and a sequence of ESAs. There are associated semantics to take into account during implementation, in that the same element can appear multiple times at different positions of the sequence. In particular, none of CSA structure fields (including HashAlgo, LocalKeyID, and AuthKeyOctets) alone or in a combination has to be unique within a given CSA, or within a given sequence of CSAs, or within all sequences of CSAs of a Babel speaker.

In the CSA space defined this way, for any two authentication keys their one field (in)equality would not imply their another field (in)equality. In other words, it is acceptable to have more than one authentication key with the same LocalKeyID or the same AuthKeyOctets or both at a time. It is a conscious design decision that CSA

semantics allow for duplication of security associations. Consequently, ESA semantics allow for duplication of intermediate ESAs in the sequence until the explicit deduplication (Section 5.2 item 4).

One of the intentions of this is to define the security association management in a way that allows the addressing of some specifics of Babel as a mesh routing protocol. For example, a system operator configuring a Babel speaker to participate in more than one administrative domain could find each domain using its own authentication key (AuthKeyOctets) under the same LocalKeyID value, e.g., a "well-known" or "default" value like 0 or 1. Since reconfiguring the domains to use distinct LocalKeyID values isn't always feasible, the multi-domain Babel speaker using several distinct authentication keys under the same LocalKeyID would make a valid use case for such duplication.

Furthermore, if in this situation the operator decided to migrate one of the domains to a different LocalKeyID value in a seamless way, respective Babel speakers would use the same authentication key (AuthKeyOctets) under two different LocalKeyID values for the time of the transition (see also item (e) of Section 9). This would make a similar use case.

Another intention of this design decision is to decouple security association management from authentication key management as much as possible, so that the latter, be it manual keying or a key management protocol, could be designed and implemented independently. This way the additional key management constraints, if any, would be left out of scope of this authentication mechanism. A similar thinking justifies LocalKeyID field having bit length in ESA structure definition, but not in that of CSA.

7. Network Management Aspects

7.1. Backward Compatibility

Support of this mechanism is optional, it does not change the default behaviour of a Babel speaker and causes no compatibility issues with speakers properly implementing the original Babel specification. Given two Babel speakers, one implementing this mechanism and configured for authenticated exchange (A) and another not implementing it (B), these would not distribute routing information uni-directionally or form a routing loop or experience other protocol logic issues specific purely to the use of this mechanism.

The Babel design requires a bi-directional neighbour reachability

condition between two given speakers for a successful exchange of routing information. Apparently, in the case above neighbour reachability would be uni-directional. Presence of TS/PC and HMAC TLVs in Babel packets sent by A would be transparent to B. But lack of authentication data in Babel packets send by B would make them effectively invisible to the instance of the original protocol of A. Uni-directional links are not specific to use of this mechanism, they naturally exist on their own and are properly detected and coped with by the original protocol (see Section 3.4.2 of [BABEL]).

7.2. Multi-Domain Authentication

The receiving procedure treats a packet as authentic as soon as one of its HMAC TLVs passes the check against the derived sequence of ESAs. This allows for packet exchange authenticated with multiple (hash algorithm, authentication key) pairs simultaneously, in combinations as arbitrary as permitted by MaxDigestsIn and MaxDigestsOut.

For example, consider three Babel speakers with one interface each, configured with the following CSAs:

- speaker A: (hash algorithm H1; key SK1), (hash algorithm H1; key SK2)
- speaker B: (hash algorithm H1; key SK1)
- speaker C: (hash algorithm H1; key SK2)

Packets sent by A would contain 2 HMAC TLVs each, packets sent by B and C would contain 1 HMAC TLV each. A and B would authenticate the exchange between themselves using H1 and SK1; A and C would use H1 and SK2; B and C would discard each other’s packets.

Consider a similar set of speakers configured with different CSAs:

- speaker D: (hash algorithm H2; key SK3), (hash algorithm H3; key SK4)
- speaker E: (hash algorithm H2; key SK3), (hash algorithm H4, keys SK5 and SK6)
- speaker F: (hash algorithm H3; keys SK4 and SK7), (hash algorithm H5, key SK8)

Packets sent by D would contain 2 HMAC TLVs each, packets sent by E and F would contain 3 HMAC TLVs each. D and E would authenticate the exchange between themselves using H2 and SK3; D and F would use H3

and SK4; E and F would discard each other's packets. The simultaneous use of H4, SK5, and SK6 by E, as well as use of SK7, H5, and SK8 by F (for their own purposes) would remain insignificant to A.

An operator implementing a multi-domain authentication should keep in mind that values of MaxDigestsIn and MaxDigestsOut may be different both within the same Babel speaker and across different speakers. Since the minimum value of both parameters is 2 (see Section 3.4 and Section 3.5), when more than 2 authentication domains are configured simultaneously it is advised to confirm that every involved speaker can handle sufficient number of HMAC results for both sending and receiving.

The recommended method of Babel speaker configuration for multi-domain authentication is not only using a different authentication key for each domain, but also using a separate CSA for each domain, even when hash algorithms are the same. This allows for fair competition between CSAs and sometimes limits the consequences of a possible misconfiguration to the scope of one CSA. See also item (e) of Section 9.

7.3. Migration to and from Authenticated Exchange

It is common in practice to consider a migration to authenticated exchange of routing information only after the network has already been deployed and put to an active use. Performing the migration in a way without regular traffic interruption is typically demanded, and this specification allows a smooth migration using the RxAuthRequired interface parameter defined in Section 3.1. This measure is similar to the "transition mode" suggested in Section 5 of [OSPF3-AUTH].

An operator performing the migration needs to arrange configuration changes as follows:

1. Decide on particular hash algorithm(s) and key(s) to be used.

2. Identify all speakers and their involved interfaces that need to be migrated to authenticated exchange.

3. For each of the speakers and the interfaces to be reconfigured first set RxAuthRequired parameter to FALSE, then configure necessary CSA(s).

4. Examine the speakers to confirm that Babel packets are successfully authenticated according to the configuration (supposedly, through examining ANM table entries and authentication-specific statistics, see Figure 1) and address any

discrepancies before proceeding further.

5. For each of the speakers and the reconfigured interfaces set the RxAuthRequired parameter to TRUE.

Likewise, temporarily setting RxAuthRequired to FALSE can be used to migrate smoothly from an authenticated packet exchange back to unauthenticated one.

7.4. Handling of Authentication Keys Exhaustion

This specification employs a common concept of multiple authenticaion keys co-existing for a given interface, with two independent lifetime ranges associated with each key (one for sending and another for receiving). It is typically recommended to configure the keys using finite lifetimes, adding new keys before the old keys expire. However, it is obviously possible for all keys to expire for a given interface (for sending or receiving or both). Possible ways of addressing this situation raise their own concerns:

- Automatic switching to unauthenticated protocol exchange. This behaviour invalidates the initial purposes of authentication and is commonly viewed as "unacceptable" ([RIP2-AUTH] Section 5.1, [OSPF2-AUTH] Section 3.2, [OSPF3-AUTH] Section 3).

- Stopping routing information exchange over the interface. This behaviour is likely to impact regular traffic routing and is commonly viewed as "not advisable" (ibid.).

- Use of the "most recently expired" key over its intended lifetime range. This behaviour is commonly recommended for implementation (ibid.), although it may become a problem due to an offline cryptographic attack (see item (e) of Section 9) or a compromise of the key. In addition, telling a recently expired key from a key never ever been in a use may be impossible after a router restart.

Design of this mechanism prevents the automatic switching to unauthenticated exchange and is consistent with similar authentication mechanisms in this regard. But since the best choice between two other options depends on local site policy, this decision is left up to the operator rather than the implementor (in a way resembling the "fail secure" configuration knob described in Section 5.1 of [RIP2-AUTH]).

Although the deriving procedure does not allow for any exceptions in expired keys filtering (Section 5.2 item 2), the operator can trivially enforce one of the two remaining behaviour options through

local key management procedures. In particular, when using the key over its intended lifetime is more preferred than regular traffic disruption, the operator would explicitly leave the old key expiry time open until the new key is added to the router configuration. In the opposite case the operator would always configure the old key with a finite lifetime and bear associated risks.

8. Implementation Status

[RFC Editor: before publication please remove this section and the reference to [I-D.sheffer-running-code], along the offered experiment of which this section exists to assist document reviewers.]

At the time of this writing the original Babel protocol is available in two free, production-quality implementations:

- The "standalone" babeld, a BSD-licensed software with source code available on GitHub [1].

  That implementation does not support this authentication mechanism.

- The integrated babeld component of Quagga-RE, a work derived from Quagga routing protocol suite, a GPL-lisensed software with source code available on GitHub [2].

  That implementation supports this authentication mechanism as defined in revision 02 of this document. It supports both mandatory-to-implement hash algorithms (SHA-512 and Whirlpool) and a few additional algorithms (RIPEMD-160, SHA-224, SHA-256, and SHA-384). It does not support more than one link-local IPv6 address per interface. It implements authentication-specific parameters, data structures and methods as follows (whether a parameter can be "changed at runtime", it is done by means of CLI and can also be set in a configuration file):

  * MaxDigestsIn value is fixed to 4.

  * MaxDigestsOut value is fixed to 4.

  * RxAuthRequired value is specific to each interface and can be changed at runtime.

  * ANM Table contents is not retained across speaker restarts, can be retrieved and reset (all entries at once) by means of CLI.

* ANM Timeout value is specific to the whole protocol instance, has a default value of 300 seconds and can be changed at runtime.

* Ordering of elements within each interface's sequence of CSAs is arbitrary as set by operator at runtime. CSAs are implemented to refer to existing key chain syntax items. Elements of an interface's sequence of CSAs are constrained to be unique reference-wise, but not contents-wise, that is, it is possible to duplicate security associations using a different key chain name to contain the same keys.

* Ordering of elements within each KeyChain sequence is fixed to the sort order of LocalKeyID. LocalKeyID is constrained to be unique within each KeyChain sequence.

* TS/PC number updates method can be configured at runtime for the whole protocol instance to one of two methods standing for items (a) and (b) of Section 5.1. The default method is (b).

* Most of the authentication-specific statistics counters listed in Section 5.5 are implemented (per protocol instance and per each interface) and their readings are available by means of CLI with an option to log respective events into a file.

No other implementations of this authentication mechanism are known to exist, thus interoperability can only be assessed on paper. The only existing implementation has been tested to be fully compatible with itself.

9. Security Considerations

Use of this mechanism implies requirements common to a use of shared authentication keys, including, but not limited to:

- holding the keys secret,
- including sufficient amounts of random bits into each key,
- rekeying on a regular basis, and
- never reusing a used key for a different purpose

That said, proper design and implementation of a key management policy is out of scope of this work. Many publications on this subject exist and should be used for this purpose.

Considering particular attacks being in-scope or out of scope on one hand and measures taken to protect against particular in-scope attacks on the other, the original Babel protocol and this authentication mechanism are in line with similar datagram-based routing protocols and their respective mechanisms. In particular, the primary concerns addressed are:

a. Peer Entity Authentication

The Babel speaker authentication mechanism defined herein is believed to be as strong as is the class itself that it belongs to. This specification is built on fundamental concepts implemented for authentication of similar routing protocols: per-packet authentication, use of HMAC construct, use of shared keys. Although this design approach does not address all possible concerns, it is so far known to be sufficient for most practical cases.

b. Data Integrity

Meaningful parts of a Babel datagram are the contents of the Babel packet (in the definition of Section 4.2 of [BABEL]) and IPv6 source address of the datagram (Section 3.5.3 ibid.). This mechanism authenticates both parts using an HMAC construct, so that making any meaningful change to an authenticated packet after it has been emitted by the sender should be as hard as attacking the hash algorithm itself or successfully recovering the authentication key.

Note well that any trailing data of the Babel datagram is not meaningful in the scope of the original specification and does not belong to the Babel packet. Integrity of the trailing data is respectively not protected by this mechanism. At the same time, although any TLV extra data is also not meaningful in the same scope, its integrity is protected, since this extra data is a part of the Babel packet (see Figure 2).

c. Replay Attacks

This specification establishes a basic replay protection measure (see Section 3.6), defines a timeout parameter affecting its strength (see Section 3.7), and outlines implementation methods also affecting protection strength in several ways (see Section 5.1). The implementor’s choice of the timeout value and particular implementation methods may be suboptimal due to, for example, insufficient hardware resources of the Babel speaker. Furthermore, it may be possible that an operator configures the timeout and the methods to address particular local specifics and

this further weakens the protection. An operator concerned about replay attack protection strength should understand these factors and their meaning in a given network segment.

d. Denial of Service

Proper deployment of this mechanism in a Babel network significantly increases the efforts required for an attacker to feed arbitrary Babel PDUs into protocol exchange (with an intent of attacking a particular Babel speaker or disrupting exchange of regular traffic in a routing domain). It also protects the neighbour table from being flooded with forged speaker entries.

At the same time, this protection comes with a price of CPU time being spent on HMAC computations. This may be a concern for low-performance CPUs combined with high-speed interfaces, as sometimes seen in embedded systems and hardware routers. The MaxDigestsIn parameter, which is used to limit the maximum amount of CPU time spent on a single received Babel packet, addresses this concern to some extent.

The following in-scope concerns are not addressed:

e. Offline Cryptographic Attacks

This mechanism is obviously subject to offline cryptographic attacks. As soon as an attacker has obtained a copy of an authenticated Babel packet of interest (which gets easier to do in wireless networks), he has got all the parameters of the authentication-specific processing performed by the sender, except authentication key(s) and choice of particular hash algorithm(s). Since digest lengths of common hash algorithms are well-known and can be matched with those seen in the packet, complexity of this attack is essentially that of the authentication key attack.

Viewing the cryptographic strength of particular hash algorithms as a concern of its own, the main practical means of resisting offline cryptographic attacks on this mechanism are periodic rekeying and use of strong keys with a sufficient number of random bits.

It is important to understand that in the case of multiple keys being used within single interface (for a multi-domain authentication or during a key rollover) the strength of the combined configuration would be that of the weakest key, since only one successful HMAC test is required for an authentic packet. Operators concerned about offline cryptographic attacks

should enforce the same strength policy for all keys used for a given interface.

Note that a special pathological case is possible with this mechanism. Whenever two or more authentication keys are configured for a given interface such that all keys share the same AuthKeyOctets and the same HashAlgo, but LocalKeyID modulo 2^16 is different for each key, these keys will not be treated as duplicate (Section 5.2 item 4), but an HMAC result computed for a given packet will be the same for each of these keys. In the case of sending procedure this can produce multiple HMAC TLVs with exactly the same value of the Digest field, but different values of KeyID field. In this case the attacker will see that the keys are the same, even without the knowledge of the key itself. Reuse of authentication keys is not the intended use case of this mechanism and should be strongly avoided.

f. Non-repudiation

This specification relies on a use of shared keys. There is no timestamp infrastructure and no key revocation mechanism defined to address a shared key compromise. Establishing the time that a particular authentic Babel packet was generated is thus not possible. Proving that a particular Babel speaker had actually sent a given authentic packet is also impossible as soon as the shared key is claimed compromised. Even with the shared key not being compromised, reliably identifying the speaker that had actually sent a given authentic Babel packet is not possible any better than proving the speaker belongs to the group sharing the key (any of the speakers sharing a key can impose any other speaker sharing the same key).

g. Confidentiality Violations

The original Babel protocol does not encrypt any of the information contained in its packets. The contents of a Babel packet is trivial to decode, revealing network topology details. This mechanism does not improve this situation in any way. Since routing protocol messages are not the only kind of information subject to confidentiality concerns, a complete solution to this problem is likely to include measures based on the channel security model, such as IPSec and WPA2 at the time of this writing.

h. Key Management

Any authentication key exchange/distribution concerns are left out of scope. However, the internal representation of

authentication keys (see Section 3.8) allows for diverse key management means, manual configuration in the first place.

i. Message Deletion

Any message deletion attacks are left out of scope. Since a datagram deleted by an attacker cannot be distinguished from a datagram naturally lost in transmission and since datagram-based routing protocols are designed to withstand a certain loss of packets, the currently established practice is treating authentication purely as a per-packet function without any added detection of lost packets.

10. IANA Considerations

[RFC Editor: please do not remove this section.]

At the time of this publication Babel TLV Types namespace did not have an IANA registry. TLV types 11 and 12 were assigned (see Table 1) to the TS/PC and HMAC TLV types by Juliusz Chroboczek, designer of the original Babel protocol. Therefore, this document has no IANA actions.

11. Acknowledgements

Thanks to Randall Atkinson and Matthew Fanto for their comprehensive work on [RIP2-AUTH] that initiated a series of publications on routing protocols authentication, including this one. This specification adopts many concepts belonging to the whole series.

Thanks to Juliusz Chroboczek for his works on mesh networking in general and the Babel routing protocol in particular, and also for feedback on early revisions of this document. This work would not be possible without prior works on Babel. Thanks to Gabriel Kerneis and Dominic Mulligan for reviewing and proofreading early revisions of this document. Thanks to Riku Hietamaki for suggesting the test vectors section.

Thanks to Jim Gettys and Dave Taht for developing CeroWrt wireless router project and collaborating on many integration issues. A practical need for Babel authentication emerged during a research based on CeroWrt that eventually became the very first use case of this mechanism.

Thanks to Kunihiro Ishiguro and Paul Jakma for establishing GNU Zebra and Quagga routing software projects respectively. Thanks to Werner

Koch, the author of Libgcrypt. The very first implementation of this mechanism was made on base of Quagga and Libgcrypt.

This document was produced using the xml2rfc ([RFC2629]) authoring tool.

12. References

12.1. Normative References

[RFC2104] Krawczyk, H., Bellare, M., and R. Canetti, "HMAC: Keyed-Hashing for Message Authentication", RFC 2104, February 1997.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

[FIPS-198] US National Institute of Standards & Technology, "The Keyed-Hash Message Authentication Code (HMAC)", FIPS PUB 198 , March 2002.

[BABEL] Chroboczek, J., "The Babel Routing Protocol", RFC 6126, April 2011.

[I-D.sheffer-running-code] Sheffer, Y. and A. Farrel, "Improving Awareness of Running Code: the Implementation Status Section", draft-sheffer-running-code-04 (work in progress), April 2013.

12.2. Informative References

[RFC2629] Rose, M., "Writing I-Ds and RFCs using XML", RFC 2629, June 1999.

[RIP2-AUTH] Atkinson, R. and M. Fanto, "RIPv2 Cryptographic Authentication", RFC 4822, February 2007.

[OSPF2-AUTH] Bhatia, M., Manral, V., Fanto, M., White, R., Barnes, M., Li, T., and R. Atkinson, "OSPFv2 HMAC-SHA Cryptographic Authentication", RFC 5709, October 2009.

[RFC6039] Manral, V., Bhatia, M., Jaeggli, J., and R. White, "Issues with Existing Cryptographic Protection Methods for Routing

Protocols", RFC 6039, October 2010.

[OSPF3-AUTH]
Bhatia, M., Manral, V., and A. Lindem, "Supporting Authentication Trailer for OSPFv3", RFC 6506, February 2012.

[RFC6709] Carpenter, B., Aboba, B., and S. Cheshire, "Design Considerations for Protocol Extensions", RFC 6709, September 2012.

URIs

[1] <https://github.com/jech/babeld>

[2] <https://github.com/Quagga-RE/quagga-RE>

Appendix A. Figures and Tables

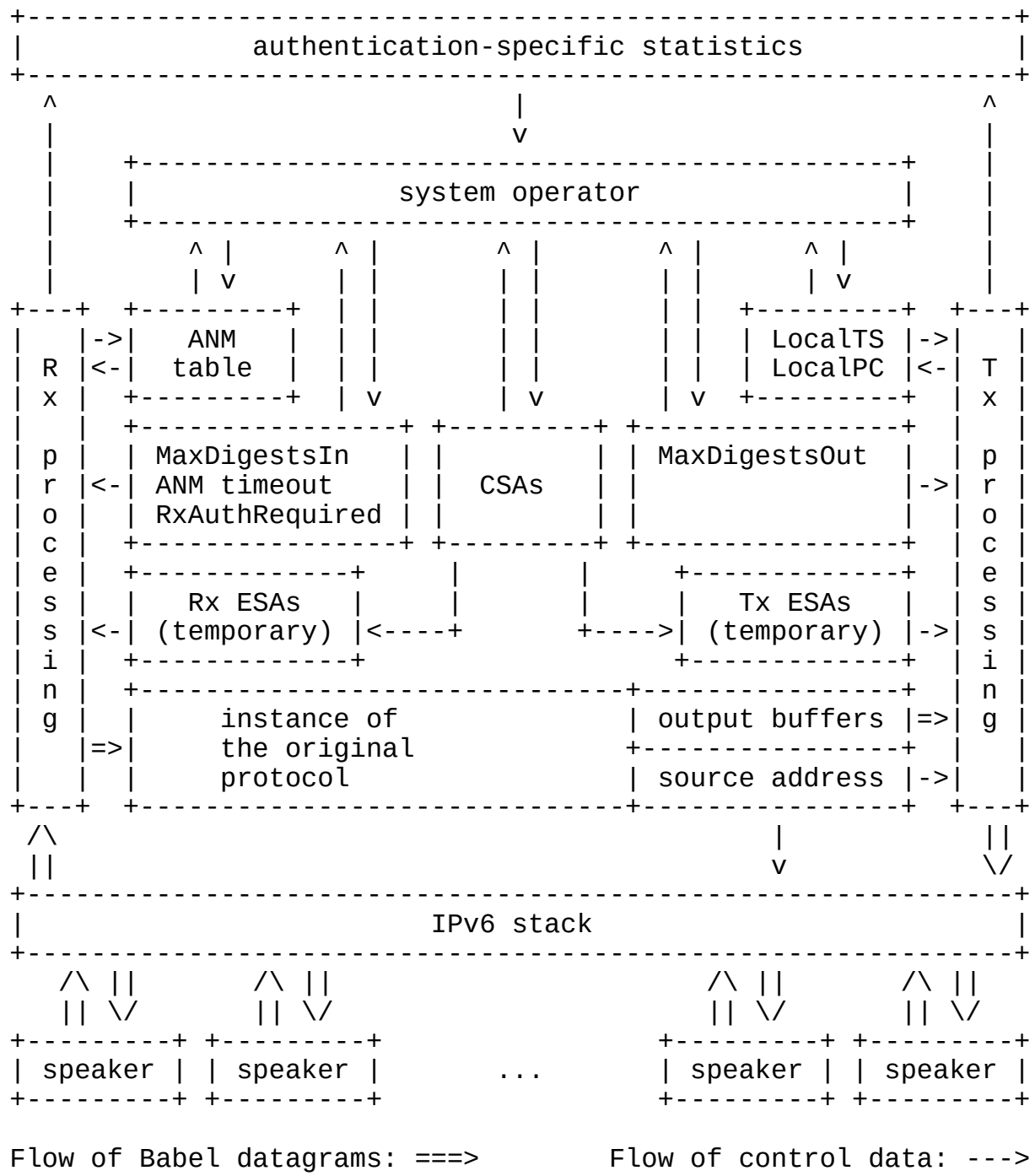


Figure 1: Interaction Diagram

The diagram below depicts structure of two Babel datagrams. The left datagram contains an unauthenticated Babel packet and an optional trailing data block. The right datagram, besides these, contains authentication-specific TLVs in the Babel packet body.

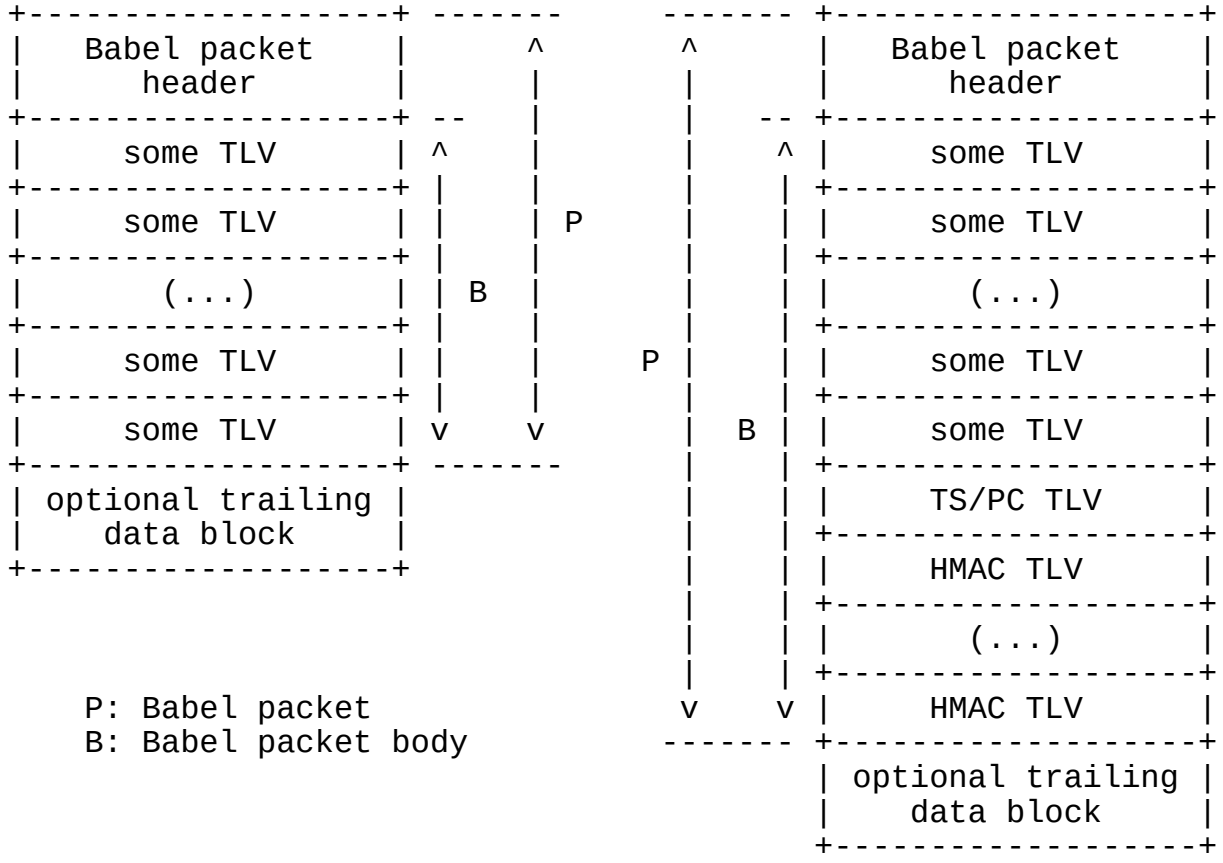


Figure 2: Babel Datagram Structure

Value	Code	Reference
0	Pad1	[BABEL]
1	PadN	[BABEL]
2	Acknowledgement Request	[BABEL]
3	Acknowledgement	[BABEL]
4	Hello	[BABEL]
5	IHU	[BABEL]
6	Router-Id	[BABEL]
7	Next Hop	[BABEL]
8	Update	[BABEL]
9	Route Request	[BABEL]
10	Seqno Request	[BABEL]
11	TS/PC	this document
12	HMAC	this document

Table 1: Babel TLV Types Namespace

Packet field	Packet octets (hexadecimal)	Meaning (decimal)
Magic	2a	42
Version	02	version 2
Body length	00:14	20 octets
[TLV] Type	04	4 (Hello)
[TLV] Length	06	6 octets
Reserved	00:00	no meaning
Seqno	7d:60	32096
Interval	01:90	400
[TLV] Type	08	8 (Update)
[TLV] Length	0a	10 octets
AE	00	0 (wildcard)
Flags	40	default router-id
Plen	00	0 bits
Omitted	00	0 bits
Interval	ff:ff	infinity
Seqno	7c:88	31880
Metric	ff:ff	infinity

Table 2: A Babel Packet without Authentication TLVs

Packet field	Packet octets (hexadecimal)	Meaning (decimal)
Magic	2a	42
Version	02	version 2
Body length	00:a4	164 octets
[TLV] Type	04	4 (Hello)
[TLV] Length	06	6 octets
Reserved	00:00	no meaning
Seqno	7d:60	32096
Interval	01:90	400
[TLV] Type	08	8 (Update)
[TLV] Length	0a	10 octets
AE	00	0 (wildcard)
Flags	40	default router-id
Plen	00	0 bits
Omitted	00	0 bits
Interval	ff:ff	infinity
Seqno	7c:88	31880
Metric	ff:ff	infinity
[TLV] Type	0b	11 (TS/PC)
[TLV] Length	06	6 octets
PacketCounter	00:01	1
Timestamp	51:5a:68:ee	1364879598
[TLV] Type	0c	12 (HMAC)
[TLV] Length	42	66 octets
KeyID	00:c8	200
Digest	fe:80:00:00:00:00:00:00:0a:11	padding
	96:ff:fe:1c:10:c8:00:00:00:00	
	00:00:00:00:00:00:00:00:00:00	
	00:00:00:00:00:00:00:00:00:00	
	00:00:00:00:00:00:00:00:00:00	
	00:00:00:00:00:00:00:00:00:00	
	00:00:00:00	
[TLV] Type	0c	12 (HMAC)
[TLV] Length	42	66 octets
KeyID	00:64	100
Digest	fe:80:00:00:00:00:00:00:0a:11	padding
	96:ff:fe:1c:10:c8:00:00:00:00	
	00:00:00:00:00:00:00:00:00:00	
	00:00:00:00:00:00:00:00:00:00	
	00:00:00:00:00:00:00:00:00:00	
	00:00:00:00:00:00:00:00:00:00	
	00:00:00:00	

Table 3: A Babel Packet with Each HMAC TLV Padded Using IPv6 Address fe80::0a11:96ff:fe1c:10c8

Packet field	Packet octets (hexadecimal)	Meaning (decimal)
Magic	2a	42
Version	02	version 2
Body length	00:a4	164 octets
[TLV] Type	04	4 (Hello)
[TLV] Length	06	6 octets
Reserved	00:00	no meaning
Seqno	7d:60	32096
Interval	01:90	400
[TLV] Type	08	8 (Update)
[TLV] Length	0a	10 octets
AE	00	0 (wildcard)
Flags	40	default router-id
Plen	00	0 bits
Omitted	00	0 bits
Interval	ff:ff	infinity
Seqno	7c:88	31880
Metric	ff:ff	infinity
[TLV] Type	0b	11 (TS/PC)
[TLV] Length	06	6 octets
PacketCounter	00:01	1
Timestamp	51:5a:68:ee	1364879598
[TLV] Type	0c	12 (HMAC)
[TLV] Length	42	66 octets
KeyID	00:c8	200
Digest	4c:72:34:27:23:1a:9a:26:71:0c 6b:24:40:58:cb:cb:e5:a6:c2:80 9d:31:13:00:3c:a3:52:0d:c6:07 13:69:0a:6e:32:84:44:b6:97:8b 0d:85:e6:8f:80:d1:ec:c0:dc:db 28:c2:15:42:51:36:04:15:3b:37 7f:3d:e1:72	HMAC result
[TLV] Type	0c	12 (HMAC)
[TLV] Length	42	66 octets
KeyID	00:64	100
Digest	ea:b3:e0:80:18:70:1a:a3:9c:d7 cf:1d:dd:06:51:5d:e6:ab:02:99 82:2f:cd:b5:a4:b6:f0:c6:a9:fc 04:50:1b:bd:82:4d:0d:28:90:a8 90:32:dc:f6:5e:ad:7c:74:c2:68 0c:8a:89:2a:bb:9e:09:ae:b0:a6 60:98:5d:9b	HMAC result

Table 4: A Babel Packet with Each HMAC TLV Containing an HMAC Result

Appendix B. Test Vectors

The test vectors below may be used to verify the correctness of some procedures performed by an implementation of this mechanism, namely:

- appending of TS/PC and HMAC TLVs to the Babel packet body,
- padding of the HMAC TLV(s),
- computing of the HMAC result(s), and
- placement of the result(s) in the TLV(s).

This verification isn't exhaustive, there are other implementation aspects that would require testing methods of their own.

The test vectors were produced as follows.

1. A Babel speaker with a network interface with IPv6 link-local address fe80::0a11:96ff:fe1c:10c8 was configured to use two CSAs for the interface:
   - CSA1={HashAlgo=SHA-512, KeyChain={{LocalKeyID=200, AuthKeyOctets=Key70}}}
   - CSA2={HashAlgo=Whirlpool, KeyChain={{LocalKeyId=100, AuthKeyOctets=Key26}}}

   The authentication keys above are:

   - Key70 in ASCII:

```
This=key=is=exactly=70=octets=long.=ABCDEFGHIJKLMNOPQRSTUVWXYZ01234567
```

   - Key70 in hexadecimal:

```
54:68:69:73:3d:6b:65:79:3d:69:73:3d:65:78:61:63
74:6c:79:3d:37:30:3d:6f:63:74:65:74:73:3d:6c:6f
6e:67:2e:3d:41:42:43:44:45:46:47:48:49:4a:4b:4c
4d:4e:4f:50:51:52:53:54:55:56:57:58:59:5a:30:31
32:33:34:35:36:37
```

   - Key26 in ASCII:

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

* Key26 in hexadecimal:

```
41:42:43:44:45:46:47:48:49:4a:4b:4c:4d:4e:4f:50
51:52:53:54:55:56:57:58:59:5a
```

KeyStartAccept, KeyStopAccept, KeyStartGenerate and KeyStopGenerate were set to make both authentication keys valid. The length of each key was picked to relate (in the terms of Section 2.4) with the properties of respective hash algorithm as follows:

* Key70 is 70 octets long. The digest length (L) of SHA-512 is 64 octets. The internal block size (B) of SHA-512 is 128 octets.

* Key26 is 26 octets long. The digest length (L) of Whirlpool is 64 octets. The internal block size (B) of Whirlpool is 64 octets.

2. The instance of the original protocol of the speaker produced a Babel packet (PktO) to be sent from the interface. Table 2 provides a decoding of PktO, contents of which is below:

```
2a:02:00:14:04:06:00:00:7d:60:01:90:08:0a:00:40
00:00:ff:ff:7c:88:ff:ff
```

3. The authentication mechanism appended one TS/PC TLV and two HMAC TLVs to the packet body, updated the "Body length" packet header field and padded the Digest field of the HMAC TLVs using the link-local IPv6 address of the interface and necessary amount of zeroes. Table 3 provides a decoding of the resulting temporary packet (PktT), contents of which is below:

```
2a:02:00:a4:04:06:00:00:7d:60:01:90:08:0a:00:40
00:00:ff:ff:7c:88:ff:ff:0b:06:00:01:51:5a:68:ee
0c:42:00:c8:fe:80:00:00:00:00:00:00:0a:11:96:ff
fe:1c:10:c8:00:00:00:00:00:00:00:00:00:00:00:00
00:00:00:00:00:00:00:00:00:00:00:00:00:00:00:00
00:00:00:00:00:00:00:00:00:00:00:00:00:00:00:00
00:00:00:00:0c:42:00:64:fe:80:00:00:00:00:00:00
0a:11:96:ff:fe:1c:10:c8:00:00:00:00:00:00:00:00
00:00:00:00:00:00:00:00:00:00:00:00:00:00:00:00
00:00:00:00:00:00:00:00:00:00:00:00:00:00:00:00
00:00:00:00:00:00:00:00
```

4. The authentication mechanism produced two HMAC results, performing the computations as follows:

* For H=SHA-512, K=Key70, and Text=PktT the HMAC result is:

```
4c:72:34:27:23:1a:9a:26:71:0c:6b:24:40:58:cb:cb
e5:a6:c2:80:9d:31:13:00:3c:a3:52:0d:c6:07:13:69
0a:6e:32:84:44:b6:97:8b:0d:85:e6:8f:80:d1:ec:c0
dc:db:28:c2:15:42:51:36:04:15:3b:37:7f:3d:e1:72
```

* For H=Whirlpool, K=Key26, and Text=PktT the HMAC result is:

```
ea:b3:e0:80:18:70:1a:a3:9c:d7:cf:1d:dd:06:51:5d
e6:ab:02:99:82:2f:cd:b5:a4:b6:f0:c6:a9:fc:04:50
1b:bd:82:4d:0d:28:90:a8:90:32:dc:f6:5e:ad:7c:74
c2:68:0c:8a:89:2a:bb:9e:09:ae:b0:a6:60:98:5d:9b
```

5. The authentication mechanism placed each HMAC result into respective HMAC TLV, producing the final authenticated Babel packet (PktA), which was eventually sent from the interface. Table 4 provides a decoding of PktA, contents of which is below:

```
2a:02:00:a4:04:06:00:00:7d:60:01:90:08:0a:00:40
00:00:ff:ff:7c:88:ff:ff:0b:06:00:01:51:5a:68:ee
0c:42:00:c8:4c:72:34:27:23:1a:9a:26:71:0c:6b:24
40:58:cb:cb:e5:a6:c2:80:9d:31:13:00:3c:a3:52:0d
c6:07:13:69:0a:6e:32:84:44:b6:97:8b:0d:85:e6:8f
80:d1:ec:c0:dc:db:28:c2:15:42:51:36:04:15:3b:37
7f:3d:e1:72:0c:42:00:64:ea:b3:e0:80:18:70:1a:a3
9c:d7:cf:1d:dd:06:51:5d:e6:ab:02:99:82:2f:cd:b5
a4:b6:f0:c6:a9:fc:04:50:1b:bd:82:4d:0d:28:90:a8
90:32:dc:f6:5e:ad:7c:74:c2:68:0c:8a:89:2a:bb:9e
09:ae:b0:a6:60:98:5d:9b
```

Interpretation of this process is to be done in the view of Figure 1, differently for the sending and the receiving directions.

For the sending direction, given a Babel speaker configured using the IPv6 address and the sequence of CSAs as described above, the implementation MUST produce exactly the temporary packet PktT if the original protocol instance produces exactly the packet PktO to be sent from the interface. The HMAC results computed afterwards MUST exactly match respective results above and the final authenticated packet MUST exactly match the PktA above.

For the receiving direction, given a Babel speaker configured using the sequence of CSAs (but not the IPv6 address) as described above, the implementation MUST (assuming the TS/PC check didn’t fail) produce exactly the temporary packet PktT above if the local IPv6 stack receives through the interface exactly the packet PktA above with the source IPv6 address above. The first HMAC result computed

afterwards MUST match the first result above. The receiving procedure doesn't compute the second HMAC result in this case, but if the implementor decides to compute it anyway for the verification purpose, it MUST exactly match the second result above.

Author's Address

Denis Ovsienko
Yandex
16, Leo Tolstoy St.
Moscow, 119021
Russia

Email: infrastation@yandex.ru