Independent Submission
Request for Comments: 7298
Updates: 6126
Category: Experimental
ISSN: 2070-1721

D. Ovsienko
Yandex

July 2014

Babel Hashed Message Authentication Code (HMAC) Cryptographic Authentication

Abstract

This document describes a cryptographic authentication mechanism for the Babel routing protocol. This document updates RFC 6126. The mechanism allocates two new TLV types for the authentication data, uses Hashed Message Authentication Code (HMAC), and is both optional and backward compatible.

Status of This Memo

This document is not an Internet Standards Track specification; it is published for examination, experimental implementation, and evaluation.

This document defines an Experimental Protocol for the Internet community. This is a contribution to the RFC Series, independently of any other RFC stream. The RFC Editor has chosen to publish this document at its discretion and makes no statement about its value for implementation or deployment. Documents approved for publication by the RFC Editor are not a candidate for any level of Internet Standard; see Section 2 of RFC 5741.

Information about the current status of this document, any errata, and how to provide feedback on it may be obtained at http://www.rfc-editor.org/info/rfc7298.

Table of Contents

1. Introduction

Authentication of routing protocol exchanges is a common means of securing computer networks. The 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 operational parameters cannot be changed.

The currently established (see [RIP2-AUTH], [OSPF2-AUTH], [ISIS-AUTH-A], [RFC6039], and [OSPF3-AUTH-BIS]) approach to an 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 Hashed Message Authentication Code (HMAC) construction (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 or only those packets protecting the integrity of protocol exchange are authenticated. This mechanism authenticates all protocol packets.

Although the HMAC construction is just one of many possible approaches to cryptographic authentication of packets, this mechanism makes use of relevant prior experience by using HMAC as well, and its

solution space correlates with the solution spaces of the mechanisms above. At the same time, it allows for a future extension that treats HMAC as a particular case of a more generic mechanism. Practical experience with the mechanism defined herein should be useful in designing such a future extension.

This specification defines the use of the cryptographic sequence number in detail sufficient to make replay attack protection strength predictable. That is, an operator can tell the strength from the declared characteristics of an implementation and, if the implementation allows the changing of relevant parameters, the effect of a reconfiguration as well.

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. This specification is not novel in this regard; for example, the Layer 2 Tunneling Protocol (L2TPv3) allows for one or two results per authenticated packet ([RFC3931] Section 5.4.1), and Mobile Ad Hoc Network (MANET) protocols allow for several ([RFC7183] Section 6.1).

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 high 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 command line interface (CLI), a management protocol such as the Simple Network Management Protocol (SNMP), a means of inter-process communication 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 BCP 14 [RFC2119].

2. Cryptographic Aspects

2.1. Mandatory-to-Implement and Optional Hash Algorithms

[RFC2104] defines HMAC as a construction 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 of [RFC6709]) 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 mandatory-to-implement hash algorithms; each is unambiguously defined and generally available in multiple implementations.

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

- RIPEMD-160 (160-bit digest)
- SHA-1 (160-bit digest)

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 128 bits or more (to meet the constraint implied in Section 2.2). Implementors SHOULD consider strong, well-known hash algorithms as additional implementation options and MUST NOT consider a hash algorithm if meaningful attacks exist for it or it is commonly viewed as deprecated.

In the latter case, it is important to take into account considerations both common (such as those made in [RFC4270]) and specific to the HMAC application of the hash algorithm. For example, [RFC6151] considers MD5 collisions and concludes that new protocol designs should not use HMAC-MD5, while [RFC6194] includes a comparable analysis of SHA-1 that finds HMAC-SHA-1 secure for the same purpose.

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 digest)
- SHA-224 (224-bit digest, SHA-2 family)
- SHA-256 (256-bit digest, SHA-2 family)
- SHA-384 (384-bit digest, SHA-2 family)
- SHA-512 (512-bit digest, SHA-2 family)
- Tiger (192-bit digest)
- Whirlpool (512-bit digest, 2nd rev., 2003)

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 construction, an implementation MUST deny the use of such keys.

2.2. Definition of Padding

Many practical applications of HMAC for authentication of datagram-based network protocols (including routing protocols) involve the padding procedure, a design-specific conditioning of the message that both the sender and the receiver perform before the HMAC computation. The specific padding procedure of this mechanism addresses the following needs:

- Data Initialization

  A design that places the HMAC result(s) computed for a message inside that same message after the computation has to have previously (i.e., before the computation) allocated in that message some data unit(s) purposed specifically for those HMAC

result(s) (in this mechanism, it is the HMAC TLV(s); see Section 4.3). The padding procedure sets the respective octets of the data unit(s), in the simplest case to a fixed value known as the padding constant.

The particular value of the constant is specific to each design. For instance, in [RIP2-AUTH] as well as works derived from it ([ISIS-AUTH-B], [OSPF2-AUTH], and [OSPF3-AUTH-BIS]), the value is 0x878FE1F3. In many other designs (for instance, [RFC3315], [RFC3931], [RFC4030], [RFC4302], [RFC5176], and [ISIS-AUTH-A]), the value is 0x00.

However, the HMAC construction is defined on the basis of a cryptographic hash algorithm, that is, an algorithm meeting a particular set of requirements made for any input message. Thus, any padding constant values, whether single- or multiple-octet, as well as any other message-conditioning methods, don't affect cryptographic characteristics of the hash algorithm and the HMAC construction, respectively.

- Source Address Protection

  In the specific case of datagram-based routing protocols, the protocol packet (that is, the message being authenticated) often does not include network-layer addresses, although the source and (to a lesser extent) the destination address of the datagram may be meaningful in the scope of the protocol instance.

  In Babel, the source address may be used as a prefix next hop (see Section 3.5.3 of [BABEL]). A well-known (see Section 2.3 of [OSPF3-AUTH-BIS]) solution to the source address protection problem is to set the first respective octets of the data unit(s) above to the source address (yet setting the rest of the octets to the padding constant). This procedure adapts this solution to the specifics of Babel, which allows for the exchange of protocol packets using both IPv4 and IPv6 datagrams (see Section 4 of [BABEL]). Even though in the case of IPv6 exchange a Babel speaker currently uses only link-local source addresses (Section 3.1 of [BABEL]), this procedure protects all octets of an arbitrary given source address for the reasons of future extensibility. The procedure implies that future Babel extensions will never use an IPv4-mapped IPv6 address as a packet source address.

This procedure does not protect the destination address, which is currently considered meaningless (Section 3.1 of [BABEL]) in the same scope. A future extension that looks to add such protection would likely use a new TLV or sub-TLV to include the destination address in the protocol packet (see Section 4.1).

Description of the padding procedure:

1. Set the first 16 octets of the Digest field of the given HMAC TLV to:

   * the given source address, if it is an IPv6 address, or

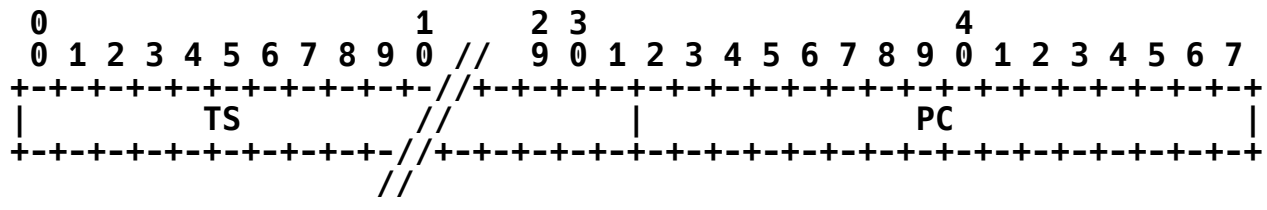
   * the IPv4-mapped IPv6 address (per Section 2.5.5.2 of [RFC4291]) holding the given source address, if it is an IPv4 address.

2. Set the remaining (TLV Length - 18) octets of the Digest field of the given HMAC TLV to 0x00 each.

For an example of a Babel packet with padded HMAC TLVs, see Table 3 in Appendix A.

2.3. Cryptographic Sequence Number Specifics

The 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 the term "TS/PC number" to avoid confusion with the route's (Section 2.5 of [BABEL]) or node's (Section 3.2.1 of [BABEL]) sequence numbers of the original Babel specification 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 a comparison of TS/PC numbers, as performed in item 3 of Section 5.4, is algebraically equivalent to a comparison of the 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 The data on which the HMAC is calculated (note item (b) of Section 8). 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 in Appendix A).

H The specific hash algorithm (see Section 2.1).

K A sequence of octets of an arbitrary, known length.

Ko The cryptographic key used with the hash algorithm.

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

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

XOR The bitwise exclusive-or operation.

Opad The hexadecimal value 0x5C repeated B times.

Ipad The hexadecimal value 0x36 repeated B times.

The algorithm below is the original, unmodified HMAC construction as defined in both [RFC2104] and [FIPS-198]; hence, it is different from the algorithms defined in [RIP2-AUTH], [ISIS-AUTH-B], [OSPF2-AUTH], and [OSPF3-AUTH-BIS] 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 the handling of keys that have length greater than L but not greater than B.

- The algorithm below does not change the value of Text before or after the computation. Padding a Babel packet before the computation and placing the result inside the packet are both 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 construction 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.

Note that in the case of Babel the Text parameter will never exceed a few thousand octets in length. In this specific case, the optimization discussed in Section 6 of [FIPS-198] applies, namely, for a given K that is more than B octets long, the following associated intermediate results may be precomputed only once: Ko, (Ko XOR Ipad), and (Ko XOR Opad).

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 Babel packet exchange to an authenticated Babel packet exchange and back (see Section 7.3). The current value of RxAuthRequired directly affects the receiving procedure defined in Section 5.4. An implementation SHOULD allow the operator to change the RxAuthRequired value at runtime or by means of a 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.

The 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 a 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.

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 a 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 its purpose, field set, and associated procedures are different.

The conceptual purpose of the ANM table is to provide longer-term replay attack protection than 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 Appendixes A and 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 the ANM timeout parameter as defined in Section 3.7) a copy of the 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 through which the authentic packet was received.

- Source

  The source address of the Babel speaker from which the authentic packet was received.

- 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 persistent memory (NVRAM) to retain the contents of the 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 be clear regarding if and how persistent memory is used for the ANM table. An implementation SHOULD allow the operator to retrieve the current contents of the ANM table at runtime. An implementation SHOULD allow the operator to remove some or all ANM table entries at runtime or by means of a 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 that stand 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.

Specifically, implementations unable to maintain the 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 the 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 that uses IPv6 for the exchange of Babel datagrams and that involves physical moves of network-interface hardware between Babel speakers. Even when performed without restarting the speakers, these physical moves 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 a 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 (hereafter referred to as "interface's sequence of CSAs"). Each interface's sequence of CSAs, as an integral part of the Babel speaker configuration, MAY be intended for 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 (hereafter referred to as "KeyChain sequence") 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), it may appear that only a fraction of the associated keys and hash

algorithms are 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 of whether this requirement is or isn't met, the implementation MUST provide a means 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 using CSAs.

The KeyChain sequence has a direct prototype, which is the "key chain" syntax item of some existing router configuration languages. If an implementation already implements this syntax item, it is suggested that the implementation reuse it, that is, implement a CSA syntax item that refers to a key chain item rather than reimplement 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 persistent storage. Element ordering within each such finite sequence (hereafter referred to as "sequence of ESAs") 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, the ESA structure is the only one not directly interfaced with the system operator (see Figure 1 in Appendix A); it is not immediately present in the protocol encoding, either. However, the ESA structure 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. The ESA structure is as meaningful for a correct implementation as the other protocol data structures.

4. Updates to Protocol Encoding

4.1. Justification

The 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 the 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 as defined in Sections 4.2 and 4.3 of [BABEL] and further discussed in [BABEL-EXTENSION]:

a. A Babel packet consists of a four-octet header followed by a packet body, that is, a sequence of TLVs (see Figure 2 in Appendix A). 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 uses 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. [BABEL] allocates TLV types 0 through 10 (see Table 1 in Appendix A), defines the TLV body structure for each, and establishes the requirement for a Babel protocol instance to ignore any unknown TLV types silently. This makes it possible to examine a packet body (to validate the framing and/or to pick particular TLVs for further processing), taking into account only the type (to distinguish between a Pad1 TLV and any other TLV) and the length of each TLV, regardless of whether any additional TLV types are eventually deployed (and if so, how many).

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 the PadN TLV) cannot be extended with the extra data.

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

- The 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.

- The 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; this would add to the complexity of both specification and implementation and would increase 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 the multiplexing of data items belonging to different extensions. Such a definition is out of the scope of this work.

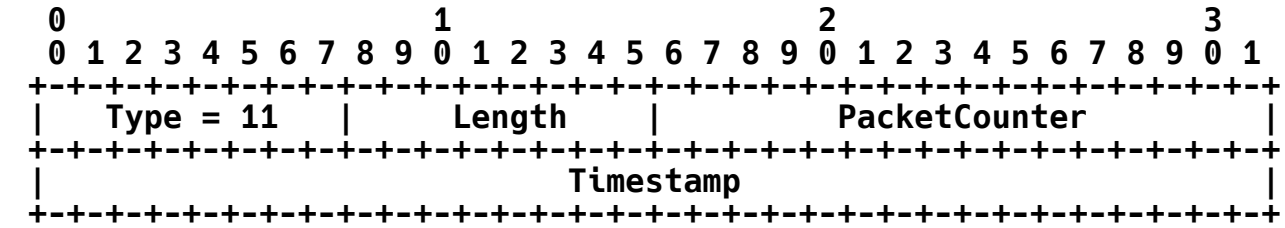
- Deprecating an extension (or only its protocol encoding) that uses purely purpose-allocated TLVs is as simple as deprecating the TLVs.

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

Considering all of the above, this mechanism uses neither the packet trailing data nor 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 in Appendix A).

4.2. TS/PC TLV

The purpose of a TS/PC TLV is to store a single TS/PC number. There is exactly one TS/PC TLV in an authenticated Babel packet.



Fields:

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

Length — The length, in octets, 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 the opposite of the ordering of TS and PC in the TS/PC number and the 48-bit equivalent (see Section 2.3).

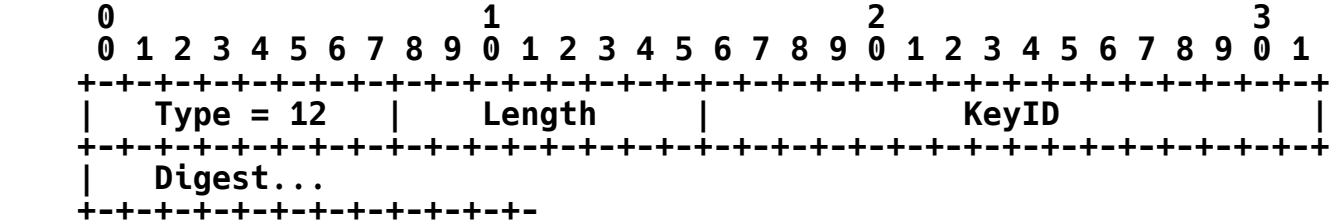
Considering the expected length and the extra data as mentioned in Section 4.3 of [BABEL], the expected length of a TS/PC TLV body is unambiguously defined as 6 octets. The receiving procedure would correctly process any TS/PC TLV with body length not less than the expected length, ignoring any extra data (Section 5.4 items 3 and 9).

The sending procedure produces a TS/PC TLV with body length equal to the expected length and the Length field, respectively, set as described in 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 adjustments to the Length TLV field, the "Body length" packet header field, and output buffer management (as explained in Section 6.2) necessary.

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 at least one HMAC TLV in an authenticated Babel packet.



Fields:

Type — Set to 12 to indicate an HMAC TLV.

Length — The length, in octets, 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 that is at least 16 octets long (see Section 2.2).

Considering the expected length and the extra data as mentioned in Section 4.3 of [BABEL], the expected length of an HMAC TLV body is not defined. The receiving and padding procedures process every octet of the Digest field, deriving the field boundary from the Length field value (Section 5.4 item 7 and Section 2.2, respectively). The sending procedure produces HMAC TLVs with the Length field precisely sizing the Digest field to match the 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 that is 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 the particular implementation method of these two updates, the advertised TS/PC number may possess additional properties that improve 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, and 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 Sections 3.6 and 3.7).

b. A more advanced implementation could make use of any 32-bit unsigned integer timestamp (number of time units since an arbitrary epoch), such as the UNIX timestamp, if 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, and 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 count" suggested in Section 4.1 of [OSPF3-AUTH-BIS], defining the updates as follows:

initialization	Set LocalPC to 0. If 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 to determine which methods of TS/PC number updates are available and whether 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 an 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 in Appendix A). 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 the KeyChain sequence of each CSA. The particular correlation maintained by this procedure implements a concept of fair (independent of the 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:

   3.1. When 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.

   3.2. 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.

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

   3.4. 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.

   3.5. ...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 the ordering of CSAs within the original input sequence of CSAs and the ordering of keys within each KeyChain sequence. This ordering maximizes the probability of having an equal amount of keys per original CSA in any N first elements of the resulting sequence. Possible optimizations 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 in Appendix A). After that, send the packet, regardless of whether 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 the TS/PC number of the current outgoing interface, as explained in Section 5.1.

3. Add to the packet body (see the note at the end of this section) a TS/PC TLV with fields set as follows:

   Type Set to 11.

   Length Set to 6.

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

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

Note that the current step may involve byte order conversion.

4. Derive a sequence of ESAs, using the 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, add to the packet body (see the note at the end of this section) an HMAC TLV with fields set as follows:

   Type Set to 12.

   Length Set to 2 plus the 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. Pad (see Section 2.2), using the source address of the current packet (see Section 6.1).

   As soon as there are MaxDigestsOut HMAC TLVs added 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 in Appendix A) 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.

Also note that it is possible to place the authentication-specific TLVs in the packet's sequence of TLVs in a number of different valid ways so long as there is exactly one TS/PC TLV in the sequence and the ordering of HMAC TLVs relative to each other, as produced in step 5 above, is preserved.

For example, see Figure 2 in Appendix A. The diagrams represent a Babel packet without (D1) and with (D2, D3, D4) authentication-specific TLVs. The optional trailing data block that is present in D1 is preserved in D2, D3, and D4. Indexing (1, 2, ..., n) of the HMAC TLVs means the order in which the sending procedure produced them (and, respectively, the HMAC results). In D2, the added TLVs are appended: the previously existing TLVs are followed by the TS/PC TLV, which is followed by the HMAC TLVs. In D3, the added TLVs are prepended: the TS/PC TLV is the first and is followed by the HMAC TLVs, which are followed by the previously existing TLVs. In D4, the added TLVs are intermixed with the previously existing TLVs and the TS/PC TLV is placed after the HMAC TLVs. All three packets meet the requirements above.

Implementors SHOULD use appending (D2) for adding the authentication-specific TLVs to the sequence; this is expected to result in more straightforward implementation and troubleshooting in most use cases.

5.4. Updates to Packet Receiving

Perform the following authentication-specific processing after an incoming Babel packet is received from the local network stack but before it is acted upon by the Babel protocol instance (see Figure 1 in Appendix A). The final action conceptually depends not only upon the result of the authentication-specific processing but also on the current value of the 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 exactly one 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 the Timestamp and PacketCounter values, respectively, of the 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 the 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. Pad (see Section 2.2) every HMAC TLV present in the temporary copy (not the original packet), using the source address of the original packet.

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 the digest length of HashAlgo of the ESA is equal to Length of the TLV and KeyID of the ESA is equal to the 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 the 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 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.

Before performing the authentication-specific processing above, an implementation SHOULD perform those basic procedures of the original protocol that don't take any protocol actions on the contents of the packet but that will discard the packet if it is not sufficiently well formed for further processing. Although the exact composition of such procedures belongs to the scope of the original protocol, it seems reasonable to state that a packet SHOULD be discarded early, regardless of whether any authentication-specific processing is due, unless its source address conforms to Section 3.1 of [BABEL] and is not the receiving speaker's own address (see item (e) of Section 8).

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-counter 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:

a. Sending an unauthenticated Babel packet through an interface having an empty sequence of CSAs (Section 5.3 item 1).

b. Sending 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).

c. Sending an authenticated Babel packet containing both TS/PC and HMAC TLVs (Section 5.3 item 9).

d. Accepting a Babel packet received through an interface having an empty sequence of CSAs (Section 5.4 item 1).

e. Refusing a received Babel packet due to an empty derived sequence of ESAs (Section 5.4 item 4).

f. Refusing a received Babel packet that does not contain exactly one TS/PC TLV (Section 5.4 item 2).

g. Refusing a received Babel packet due to the TS/PC TLV failing the ANM table check (Section 5.4 item 3). With possible future extensions in mind, in implementations of this mechanism, this event SHOULD leave out some small amount, per current (Interface, Source, LastTS, LastPC) tuple, of the packets refused due to the Timestamp value being equal to LastTS and the PacketCounter value being equal to LastPC.

h. Refusing a received Babel packet missing any HMAC TLVs (Section 5.4 item 8).

i. Refusing a received Babel packet due to none of the processed HMAC TLVs passing the ESA check (Section 5.4 item 8).

j. Accepting a received Babel packet having both TS/PC and HMAC TLVs (Section 5.4 item 10).

k. Delivery of a refused packet to the instance of the original protocol due to the RxAuthRequired parameter being set to FALSE.

Note that the 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-counter readings SHOULD be available to the operator at runtime.

6. Implementation Notes

6.1. Source Address Selection for Sending

Section 3.1 of [BABEL] allows for the exchange of protocol datagrams, using IPv4, IPv6, or both. The source address of the datagram is a unicast (link-local in the case of IPv6) address. Within an address family used by a Babel speaker, there may be more than one address eligible for the exchange and assigned to the same network interface. The original specification considers this case out of scope and leaves it up to the speaker's network stack to select one particular address as the datagram source address, but the sending procedure requires (Section 5.3 item 5) exact knowledge of the packet source address for proper padding of HMAC TLVs.

As long as a network interface has more than one address eligible for the exchange within the same address family, the Babel speaker SHOULD internally choose one of those addresses for Babel packet sending purposes and then indicate this choice to both the sending procedure and the network stack (see Figure 1 in Appendix A). Wherever this requirement cannot be met, this limitation MUST be clearly stated in the system documentation to allow an operator to plan network address management accordingly.

6.2. Output Buffer Management

An instance of the original protocol will buffer produced TLVs until the buffer becomes full or a delay timer has expired. 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 TLVs, HMAC TLVs, and any other TLVs -- and most likely the TLVs of the original protocol -- share the same packet space (see Figure 2 in Appendix A) and, respectively, the same buffer space, a particular portion of each interface buffer needs to be reserved for one 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
R = 8  + MaxDigestsOut * (4 + Lmax)
```

St The size of a TS/PC TLV.

Sh The size of an HMAC TLV.

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

An implementation allowing for a per-interface value of MaxDigestsOut or Lmax has to account for a different value of R across different interfaces, even interfaces having the same MTU. An implementation allowing for a runtime change to the value of R (due to MaxDigestsOut or Lmax) has to take care of the TLVs already buffered by the time of the change -- specifically, when the value of R increases.

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

6.3. Optimizations of Deriving Procedure for ESAs

The following optimizations of the deriving procedure for ESAs can reduce the 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 (1) none of the active keys expire and (2) none of the inactive keys enter into operation.

An implementation optimized in 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 the sending direction, in returning more than MaxDigestsOut ESAs in the derived sequence. Note that a similar optimization would be relatively difficult in the case of the receiving direction, since the number of ESAs actually used in examining a particular received packet (not to be confused with the number of HMAC computations) depends on additional factors besides just MaxDigestsIn.

6.4. Duplication of Security Associations

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 the CSA structure fields (including HashAlgo, LocalKeyID, and AuthKeyOctets), alone or in a combination, have 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 in this way, for any two authentication keys, their one field (in)equality would not imply 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 the operator decided in this situation to migrate one of the domains to a different LocalKeyID value in a seamless way, the respective Babel speakers would use the same authentication key (AuthKeyOctets) under two different LocalKeyID values for the time of the transition (see also item (f) of Section 8). 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 (as the respective reasoning made in Section 3.1 of [RIP2-AUTH] still applies). This way, the additional key-management constraints, if any, would remain out of the scope of this authentication mechanism. A similar thinking justifies the LocalKeyID field having a bit length in an ESA structure definition, but not in that of the 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 speakers would not distribute routing information unidirectionally, form a routing loop, or experience other protocol logic issues specific purely to the use of this mechanism.

The Babel design requires a bidirectional neighbour reachability condition between two given speakers for a successful exchange of routing information. Apparently, neighbour reachability would be unidirectional in the case above. The presence of TS/PC and HMAC TLVs in Babel packets sent by A would be transparent to B, but a lack of authentication data in Babel packets sent by B would make them effectively invisible to the instance of the original protocol of A. Unidirectional links are not specific to the 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 two HMAC TLVs each. Packets sent by B and C would contain one 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 two HMAC TLVs each. Packets sent by E and F would contain three 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 the use of SK7, H5, and SK8 by F (for their own purposes), would remain insignificant to D.

An operator implementing 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 Sections 3.4 and 3.5), when more than two authentication domains are configured simultaneously it is advisable to confirm that every involved speaker can handle a sufficient number of HMAC results for both sending and receiving.

The recommended method of Babel speaker configuration for multi-domain authentication is to not only use a different authentication key for each domain but also 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 (f) of Section 8.

7.3. Migration to and from Authenticated Exchange

It is common in practice to consider a migration to the authenticated exchange of routing information only after the network has already been deployed and put into 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-BIS].

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 the 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 (for instance, by examining ANM table entries and authentication-specific statistics; see Figure 1 in Appendix A), 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 an unauthenticated one.

7.4. Handling of Authentication Key Exhaustion

This specification employs a common concept of multiple authentication keys coexisting for a given interface, with two independent lifetime ranges associated with each key (one for sending and another for receiving). It is typically recommended that the keys be configured 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, 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, and [OSPF3-AUTH-BIS] 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" ([RIP2-AUTH], [OSPF2-AUTH], and [OSPF3-AUTH]), although [OSPF3-AUTH-BIS] is different in this regard.

- The use of the "most recently expired" key over its intended lifetime range. This behaviour is recommended for implementation in [RIP2-AUTH], [OSPF2-AUTH], and [OSPF3-AUTH] but not in [OSPF3-AUTH-BIS]. Such use of this key may become a problem, due to an offline cryptographic attack (see item (f) of Section 8) or a compromise of the key. In addition, distinguishing a recently expired key from a key that has never been used may be impossible after a router restart.

The design of this mechanism prevents 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 the filtering of expired keys (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 preferable to 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. Security Considerations

The use of this mechanism implies requirements common to the 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 are out of the scope of this work. Many publications on this subject exist and should be used for this purpose (BCP 107 [RFC4107], BCP 132 [RFC4962], and [RFC6039] are suggested as starting points).

It is possible for a network that exercises rollover of authentication keys to experience accidental expiration of all the keys for a network interface, as discussed at greater length in Section 7.4. With that and the guidance of Section 5.1 of [RIP2-AUTH] in mind, in such an event the Babel speaker MUST send a "last key expired" notification to the operator (e.g., via syslog, SNMP, and/or other implementation-specific means), most likely in relation to item (b) of Section 5.5. Also, any actual occurrence of an authentication key expiration MUST cause a security event to be logged by the implementation. The log item MUST include at least a note that the authentication key has expired, the Babel routing protocol instance(s) affected, the network interface(s) affected, the LocalKeyID that is affected, and the current date/time. Operators are encouraged to check such logs as an operational security practice.

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 the class itself to which it belongs. This specification is built on fundamental concepts implemented for authentication of similar routing protocols: per-packet authentication, the use of the HMAC construction, and the 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 the source address of the datagram (Section 3.5.3 of [BABEL]). This mechanism authenticates both parts, using the HMAC construction, 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 HMAC construction 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 thus 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 in Appendix A).

c. Denial of Service

   Proper deployment of this mechanism in a Babel network significantly increases the efforts required for an attacker to feed arbitrary Babel packets into a protocol exchange (with the intent of attacking a particular Babel speaker or disrupting the 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.

d. Reflection Attacks

Given the approach discussed in item (b), the only potential reflection attack on this mechanism could be replaying exact copies of Babel packets back to the sender from the same source address. The mitigation in this case is straightforward and is discussed in Section 5.4.

The following in-scope concern is only partially addressed:

e. 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.

That said, a particular form of replay attack on this mechanism remains possible anyway. Whether there are two or more network segments using the same CSA and there is an adversary that captures Babel packets on one segment and replays on another (and vice versa, due to the bidirectional reachability requirement for neighbourship), some of the speakers on one such segment will detect the "virtual" neighbours from another and may prefer them for some destinations. This applies even more so as Babel doesn't require a common pre-configured network prefix between neighbours.

A reliable solution to this particular problem, which Section 4.5 of [RFC7186] discusses as well, is not currently known. It is recommended that the operators use distinct CSAs for distinct network segments.

The following in-scope concerns are not addressed:

f. 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 all of the parameters of the authentication-specific processing performed by the sender, except for authentication key(s) and the 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, the 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 the 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 a single interface (for 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 the sending procedure, this can produce multiple HMAC TLVs with exactly the same value of the Digest field but different values of the KeyID field. In this case, the attacker will see that the keys are the same, even without knowledge of

the key itself. The reuse of authentication keys is not the intended use case of this mechanism and should be strongly avoided.

g. Non-repudiation

This specification relies on the use of shared keys. There is no timestamp infrastructure and no key-revocation mechanism defined to address the compromise of a shared key. 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 if the shared key is not compromised, reliably identifying the speaker that had actually sent a given authentic Babel packet is not possible. Since any of the speakers sharing a key can impersonate any other speaker sharing the same key, it is only possible to prove that the speaker belongs to the group sharing the key.

h. Confidentiality Violations

The original Babel protocol does not encrypt any of the information contained in its packets. The contents of a Babel packet are trivial to decode and thus can reveal 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 Wi-Fi Protected Access 2 (WPA2) at the time of this writing.

i. Key Management

Any authentication key exchange/distribution concerns are out of scope. However, the internal representation of authentication keys (see Section 3.8) allows implementations to use such diverse key-management techniques as manual configuration, a provisioning system, a key-management protocol, or any other means that comply with this specification.

j. Message Deletion

Any message deletion attacks are 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.

9. IANA Considerations

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

10. Acknowledgements

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

Thanks to Juliusz Chroboczek, Gabriel Kerneis, and Matthieu Boutier. This document incorporates many technical and editorial corrections based on their feedback. Thanks to all contributors to Babel, because this work would not be possible without the prior works. Thanks to Dominic Mulligan for editorial proofreading of this document. Thanks to Riku Hietamaki for suggesting the test vectors section.

Thanks to Joel Halpern, Jim Schaad, Randall Atkinson, and Stephen Farrell for providing (in chronological order) valuable feedback on earlier versions of this document.

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

Thanks to Kunihiro Ishiguro and Paul Jakma for establishing the 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 a base of Quagga and Libgcrypt.

11. References

11.1. 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.

[RFC4291] Hinden, R. and S. Deering, "IP Version 6 Addressing Architecture", RFC 4291, February 2006.

[FIPS-198] National Institute of Standards and Technology, "The Keyed-Hash Message Authentication Code (HMAC)", FIPS PUB 198-1, July 2008.

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

11.2. Informative References

[RFC3315] Droms, R., Bound, J., Volz, B., Lemon, T., Perkins, C., and M. Carney, "Dynamic Host Configuration Protocol for IPv6 (DHCPv6)", RFC 3315, July 2003.

[RFC3931] Lau, J., Townsley, M., and I. Goyret, "Layer Two Tunneling Protocol - Version 3 (L2TPv3)", RFC 3931, March 2005.

[RFC4030] Stapp, M. and T. Lemon, "The Authentication Suboption for the Dynamic Host Configuration Protocol (DHCP) Relay Agent Option", RFC 4030, March 2005.

[RFC4107] Bellovin, S. and R. Housley, "Guidelines for Cryptographic Key Management", BCP 107, RFC 4107, June 2005.

[RFC4270] Hoffman, P. and B. Schneier, "Attacks on Cryptographic Hashes in Internet Protocols", RFC 4270, November 2005.

[RFC4302] Kent, S., "IP Authentication Header", RFC 4302, December 2005.

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

[RFC4962] Housley, R. and B. Aboba, "Guidance for Authentication, Authorization, and Accounting (AAA) Key Management", BCP 132, RFC 4962, July 2007.

[RFC5176] Chiba, M., Dommety, G., Eklund, M., Mitton, D., and B. Aboba, "Dynamic Authorization Extensions to Remote Authentication Dial In User Service (RADIUS)", RFC 5176, January 2008.

[ISIS-AUTH-A]
Li, T. and R. Atkinson, "IS-IS Cryptographic Authentication", RFC 5304, October 2008.

[ISIS-AUTH-B]
Bhatia, M., Manral, V., Li, T., Atkinson, R., White, R., and M. Fanto, "IS-IS Generic Cryptographic Authentication", RFC 5310, February 2009.

[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.

[RFC6151] Turner, S. and L. Chen, "Updated Security Considerations for the MD5 Message-Digest and the HMAC-MD5 Algorithms", RFC 6151, March 2011.

[RFC6194] Polk, T., Chen, L., Turner, S., and P. Hoffman, "Security Considerations for the SHA-0 and SHA-1 Message-Digest Algorithms", RFC 6194, March 2011.

[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.

[BABEL-EXTENSION]
Chroboczek, J., "Extension Mechanism for the Babel Routing Protocol", Work in Progress, June 2014.

[OSPF3-AUTH-BIS]
Bhatia, M., Manral, V., and A. Lindem, "Supporting Authentication Trailer for OSPFv3", RFC 7166, March 2014.

[RFC7183] Herberg, U., Dearlove, C., and T. Clausen, "Integrity Protection for the Neighborhood Discovery Protocol (NHDP) and Optimized Link State Routing Protocol Version 2 (OLSRv2)", RFC 7183, April 2014.

[RFC7186] Yi, J., Herberg, U., and T. Clausen, "Security Threats for the Neighborhood Discovery Protocol (NHDP)", RFC 7186, April 2014.

Appendix A. Figures and Tables

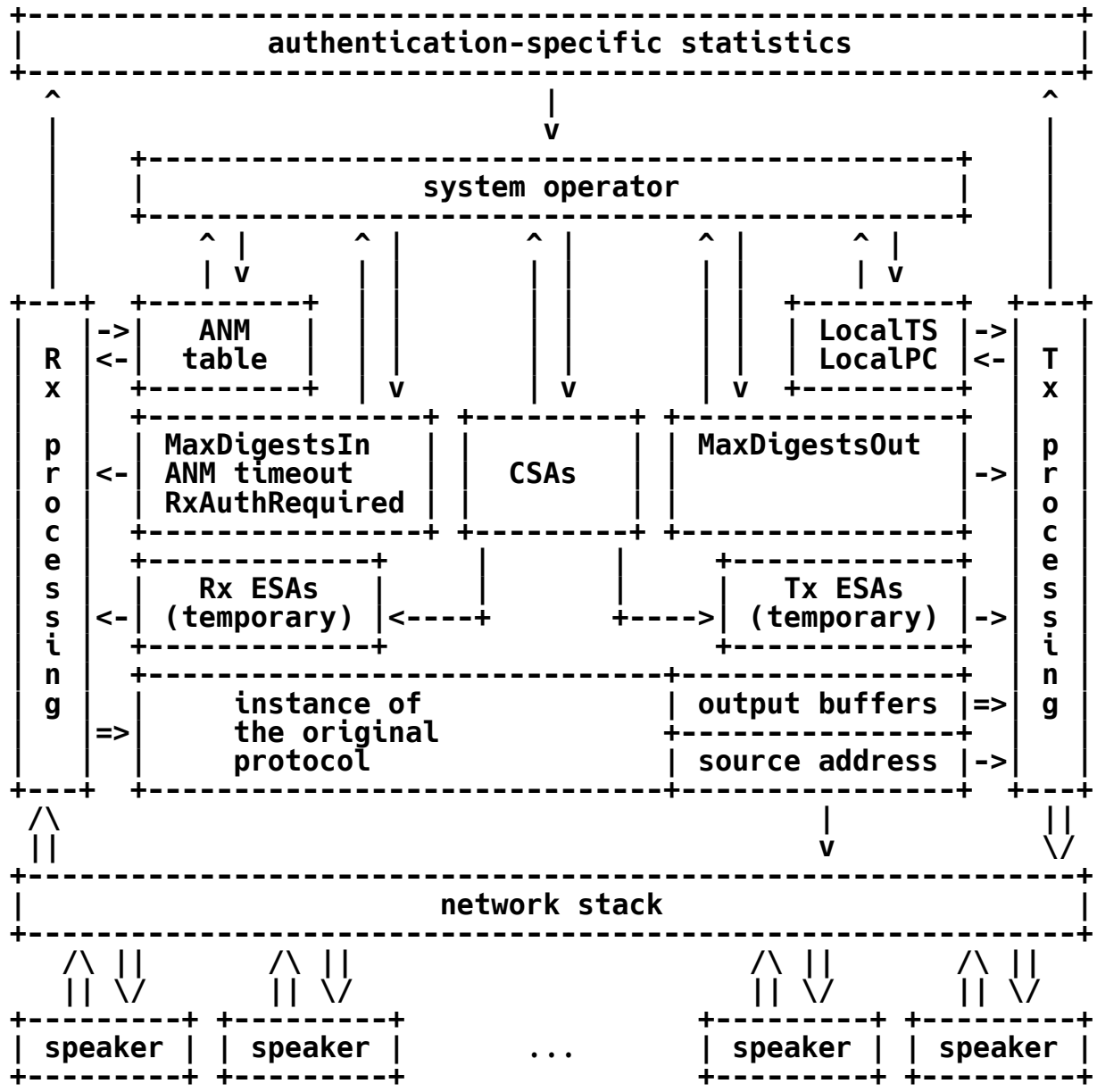


Figure 1: Interaction Diagram

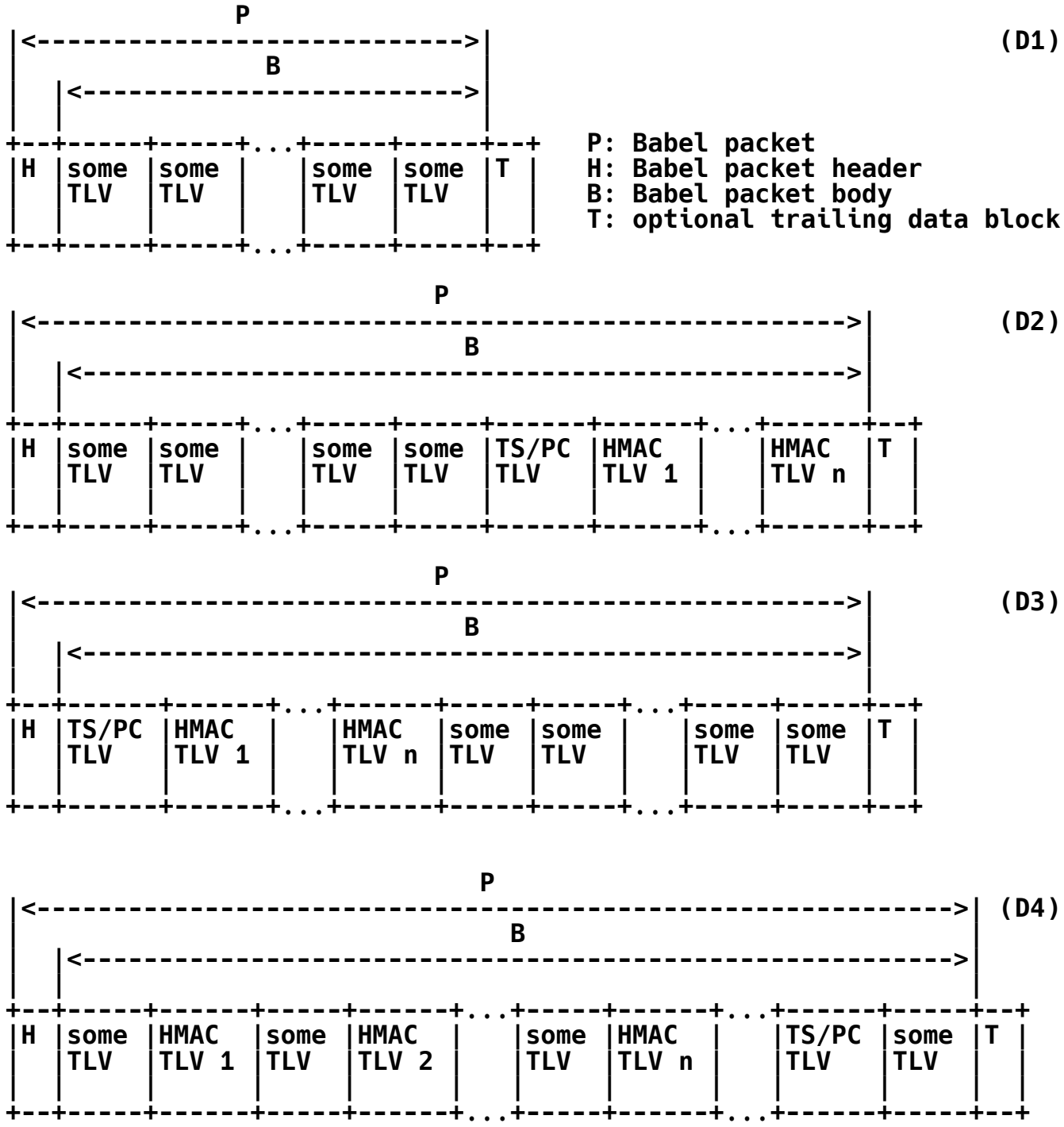


Figure 2: Babel Datagram Structure

Value	Name	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 0 through 12

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	09:25	2341
Interval	01:90	400 (4.00 s)
[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	68:21	26657
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	00:4c	76 octets
[TLV] Type	04	4 (Hello)
[TLV] Length	06	6 octets
Reserved	00:00	no meaning
Seqno	09:25	2341
Interval	01:90	400 (4.00 s)
[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	68:21	26657
Metric	ff:ff	infinity
[TLV] Type	0b	11 (TS/PC)
[TLV] Length	06	6 octets
PacketCounter	00:01	1
Timestamp	52:1d:7e:8b	1377664651
[TLV] Type	0c	12 (HMAC)
[TLV] Length	16	22 octets
KeyID	00:c8	200
Digest	fe:80:00:00:00:00:00:0a:11 96:ff:fe:1c:10:c8:00:00:00:00	padding
[TLV] Type	0c	12 (HMAC)
[TLV] Length	16	22 octets
KeyID	00:64	100
Digest	fe:80:00:00:00:00:00:0a:11 96:ff:fe:1c:10:c8:00:00:00:00	padding

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	00:4c	76 octets
[TLV] Type	04	4 (Hello)
[TLV] Length	06	6 octets
Reserved	00:00	no meaning
Seqno	09:25	2341
Interval	01:90	400 (4.00 s)
[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	68:21	26657
Metric	ff:ff	infinity
[TLV] Type	0b	11 (TS/PC)
[TLV] Length	06	6 octets
PacketCounter	00:01	1
Timestamp	52:1d:7e:8b	1377664651
[TLV] Type	0c	12 (HMAC)
[TLV] Length	16	22 octets
KeyID	00:c8	200
Digest	c6:f1:06:13:30:3c:fa:f3:eb:5d 60:3a:ed:fd:06:55:83:f7:ee:79	HMAC result
[TLV] Type	0c	12 (HMAC)
[TLV] Length	16	22 octets
KeyID	00:64	100
Digest	df:32:16:5e:d8:63:16:e5:a6:4d c7:73:e0:b5:22:82:ce:fe:e2:3c	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 TS/PC and HMAC TLVs to the Babel packet body,
- padding the HMAC TLV(s),
- computation of the HMAC result(s), and
- placement of the result(s) in the TLV(s).

This verification isn't exhaustive. There are other important 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=RIPEMD-160, KeyChain={{LocalKeyID=200, AuthKeyOctets=Key26}}}
   * CSA2={HashAlgo=SHA-1, KeyChain={{LocalKeyId=100, AuthKeyOctets=Key70}}}

   The authentication keys above are:

   * 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

   * 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

The length of each key was picked to relate (using the terms listed in Section 2.4) to the properties of its respective hash algorithm as follows:

* the digest length (L) of both RIPEMD-160 and SHA-1 is 20 octets,

* the internal block size (B) of both RIPEMD-160 and SHA-1 is 64 octets,

* the length of Key26 (26) is greater than L but less than B, and

* the length of Key70 (70) is greater than B (and thus greater than L).

KeyStartAccept, KeyStopAccept, KeyStartGenerate, and KeyStopGenerate were set to make both authentication keys valid.

2. The instance of the original protocol of the speaker produced a Babel packet (Pkt0) to be sent from the interface. Table 2 provides a decoding of Pkt0, the contents of which are below:

   2a:02:00:14:04:06:00:00:09:25:01:90:08:0a:00:40
   00:00:ff:ff:68:21: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 the necessary amount of zeroes. Table 3 provides a decoding of the resulting temporary packet (PktT), the contents of which are below:

   2a:02:00:4c:04:06:00:00:09:25:01:90:08:0a:00:40
   00:00:ff:ff:68:21:ff:ff:0b:06:00:01:52:1d:7e:8b
   0c:16:00:c8:fe:80:00:00:00:00:00:00:0a:11:96:ff
   fe:1c:10:c8:00:00:00:00:0c:16:00:64:fe:80:00:00
   00:00:00:00:0a:11:96:ff:fe:1c:10:c8:00:00:00:00

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

   * For H=RIPEMD-160, K=Key26, and Text=PktT, the HMAC result is:

     c6:f1:06:13:30:3c:fa:f3:eb:5d:60:3a:ed:fd:06:55
     83:f7:ee:79

   * For H=SHA-1, K=Key70, and Text=PktT, the HMAC result is:

     df:32:16:5e:d8:63:16:e5:a6:4d:c7:73:e0:b5:22:82
     ce:fe:e2:3c

5. The authentication mechanism placed each HMAC result into its respective HMAC TLV, producing the final authenticated Babel packet (PktA), which was eventually sent from the interface.

   Table 4 provides a decoding of PktA, the contents of which are below:

   2a:02:00:4c:04:06:00:00:09:25:01:90:08:0a:00:40
   00:00:ff:ff:68:21:ff:ff:0b:06:00:01:52:1d:7e:8b
   0c:16:00:c8:c6:f1:06:13:30:3c:fa:f3:eb:5d:60:3a
   ed:fd:06:55:83:f7:ee:79:0c:16:00:64:df:32:16:5e
   d8:63:16:e5:a6:4d:c7:73:e0:b5:22:82:ce:fe:e2:3c

Interpretation of this process is to be done differently for the sending and receiving directions (see Figure 1).

For the sending direction, given a Babel speaker configured using the IPv6 address and the sequence of CSAs as described above, the implementation SHOULD (see notes in Section 5.3) produce exactly the temporary packet PktT if the original protocol instance produces exactly the packet PktO to be sent from the interface. If the temporary packet exactly matches PktT, the HMAC results computed afterwards MUST exactly match the respective results above, and the final authenticated packet MUST exactly match PktA above.

For the receiving direction, given a Babel speaker configured using the sequence of CSAs as described above (but a different IPv6 address), the implementation MUST (assuming that the TS/PC check didn't fail) produce exactly the temporary packet PktT above if its network stack receives through the interface exactly the packet PktA above from 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 verification purposes, 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