

Network Working Group

Internet-Draft

Obsoletes: [7298](#) (if approved)

Intended status: Standards Track

Expires: December 22, 2019

C. Do

W. Kolodziejek

J. Chroboczek

IRIF, University of Paris-Diderot

June 20, 2019

**HMAC authentication for the Babel routing protocol**  
**draft-ietf-babel-hmac-06**

Abstract

This document describes a cryptographic authentication mechanism for the Babel routing protocol that has provisions for replay avoidance. This document updates RFC 6126bis and obsoletes [RFC 7298](#).

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of [BCP 78](#) and [BCP 79](#).

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at <https://datatracker.ietf.org/drafts/current/>.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on December 22, 2019.

Copyright Notice

Copyright (c) 2019 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to [BCP 78](#) and the IETF Trust's Legal Provisions Relating to IETF Documents (<https://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

## Table of Contents

<a href="#">1.</a>	Introduction . . . . .	<a href="#">2</a>
<a href="#">1.1.</a>	Applicability . . . . .	<a href="#">3</a>
<a href="#">1.2.</a>	Assumptions and security properties . . . . .	<a href="#">3</a>
<a href="#">1.3.</a>	Specification of Requirements . . . . .	<a href="#">4</a>
<a href="#">2.</a>	Conceptual overview of the protocol . . . . .	<a href="#">4</a>
<a href="#">3.</a>	Data Structures . . . . .	<a href="#">6</a>
<a href="#">3.1.</a>	The Interface Table . . . . .	<a href="#">6</a>
<a href="#">3.2.</a>	The Neighbour table . . . . .	<a href="#">6</a>
<a href="#">4.</a>	Protocol Operation . . . . .	<a href="#">7</a>
<a href="#">4.1.</a>	HMAC computation . . . . .	<a href="#">7</a>
<a href="#">4.2.</a>	Packet Transmission . . . . .	<a href="#">8</a>
<a href="#">4.3.</a>	Packet Reception . . . . .	<a href="#">8</a>
<a href="#">4.4.</a>	Expiring per-neighbour state . . . . .	<a href="#">12</a>
<a href="#">5.</a>	Packet Format . . . . .	<a href="#">12</a>
<a href="#">5.1.</a>	HMAC TLV . . . . .	<a href="#">12</a>
<a href="#">5.2.</a>	PC TLV . . . . .	<a href="#">13</a>
<a href="#">5.3.</a>	Challenge Request TLV . . . . .	<a href="#">13</a>
<a href="#">5.4.</a>	Challenge Reply TLV . . . . .	<a href="#">14</a>
<a href="#">6.</a>	Security Considerations . . . . .	<a href="#">14</a>
<a href="#">7.</a>	IANA Considerations . . . . .	<a href="#">15</a>
<a href="#">8.</a>	Acknowledgments . . . . .	<a href="#">16</a>
<a href="#">9.</a>	References . . . . .	<a href="#">16</a>
<a href="#">9.1.</a>	Normative References . . . . .	<a href="#">16</a>
<a href="#">9.2.</a>	Informational References . . . . .	<a href="#">17</a>
<a href="#">Appendix A.</a>	Incremental deployment and key rotation . . . . .	<a href="#">17</a>
<a href="#">Appendix B.</a>	Changes from previous versions . . . . .	<a href="#">18</a>
<a href="#">B.1.</a>	Changes since <a href="#">draft-ietf-babel-hmac-00</a> . . . . .	<a href="#">18</a>
<a href="#">B.2.</a>	Changes since <a href="#">draft-ietf-babel-hmac-01</a> . . . . .	<a href="#">18</a>
<a href="#">B.3.</a>	Changes since <a href="#">draft-ietf-babel-hmac-02</a> . . . . .	<a href="#">18</a>
<a href="#">B.4.</a>	Changes since <a href="#">draft-ietf-babel-hmac-03</a> . . . . .	<a href="#">18</a>
<a href="#">B.5.</a>	Changes since <a href="#">draft-ietf-babel-hmac-04</a> . . . . .	<a href="#">19</a>
<a href="#">B.6.</a>	Changes since <a href="#">draft-ietf-babel-hmac-05</a> . . . . .	<a href="#">19</a>
	Authors' Addresses . . . . .	<a href="#">19</a>

**[1.](#) Introduction**

By default, the Babel routing protocol trusts the information contained in every UDP datagram that it receives on the Babel port. An attacker can redirect traffic to itself or to a different node in the network, causing a variety of potential issues. In particular, an attacker might:

- o spoof a Babel packet, and redirect traffic by announcing a smaller metric, a larger seqno, or a longer prefix;



- o spoof a malformed packet, which could cause an insufficiently robust implementation to crash or interfere with the rest of the network;
- o replay a previously captured Babel packet, which could cause traffic to be redirected or otherwise interfere with the network.

Protecting a Babel network is challenging due to the fact that the Babel protocol uses both unicast and multicast communication. One possible approach, used notably by the Babel over Datagram Transport Layer Security (DTLS) protocol [[I-D.ietf-babel-dtls](#)], is to use unicast communication for all semantically significant communication, and then use a standard unicast security protocol to protect the Babel traffic. In this document, we take the opposite approach: we define a cryptographic extension to the Babel protocol that is able to protect both unicast and multicast traffic, and thus requires very few changes to the core protocol.

### **[1.1.](#) Applicability**

The protocol defined in this document assumes that all interfaces on a given link are equally trusted and share a small set of symmetric keys (usually just one, and two during key rotation). The protocol is inapplicable in situations where asymmetric keying is required, where the trust relationship is partial, or where large numbers of trusted keys are provisioned on a single link at the same time.

This protocol supports incremental deployment (where an insecure Babel network is made secure with no service interruption), and it supports graceful key rotation (where the set of keys is changed with no service interruption).

This protocol does not require synchronised clocks, it does not require persistently monotonic clocks, and it does not require persistent storage except for what might be required for storing cryptographic keys.

### **[1.2.](#) Assumptions and security properties**

The correctness of the protocol relies on the following assumptions:

- o that the Hashed Message Authentication Code (HMAC) being used is invulnerable to pre-image attacks, i.e., that an attacker is unable to generate a packet with a correct HMAC;
- o that a node never generates the same index or nonce twice over the lifetime of a key.



The first assumption is a property of the HMAC being used. The second assumption can be met either by using a robust random number generator [[RFC4086](#)] and sufficiently large indices and nonces, by using a reliable hardware clock, or by rekeying whenever a collision becomes likely.

If the assumptions above are met, the protocol described in this document has the following properties:

- o it is invulnerable to spoofing: any packet accepted as authentic is the exact copy of a packet originally sent by an authorised node;
- o locally to a single node, it is invulnerable to replay: if a node has previously accepted a given packet, then it will never again accept a copy of this packet or an earlier packet from the same sender;
- o among different nodes, it is only vulnerable to immediate replay: if a node A has accepted a packet from C as valid, then a node B will only accept a copy of that packet as authentic if B has accepted an older packet from C and B has received no later packet from C.

While this protocol makes serious efforts to mitigate the effects of a denial of service attack, it does not fully protect against such attacks.

### **[1.3.](#) Specification of Requirements**

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [BCP 14](#) [[RFC2119](#)] [[RFC8174](#)] when, and only when, they appear in all capitals, as shown here.

## **[2.](#) Conceptual overview of the protocol**

When a node B sends out a Babel packet through an interface that is configured for HMAC cryptographic protection, it computes one or more HMACs which it appends to the packet. When a node A receives a packet over an interface that requires HMAC cryptographic protection, it independently computes a set of HMACs and compares them to the HMACs appended to the packet; if there is no match, the packet is discarded.

In order to protect against replay, B maintains a per-interface 32-bit integer known as the "packet counter" (PC). Whenever B sends



a packet through the interface, it embeds the current value of the PC within the region of the packet that is protected by the HMACs and increases the PC by at least one. When A receives the packet, it compares the value of the PC with the one contained in the previous packet received from B, and unless it is strictly greater, the packet is discarded.

By itself, the PC mechanism is not sufficient to protect against replay. Consider a peer A that has no information about a peer B (e.g., because it has recently rebooted). Suppose that A receives a packet ostensibly from B carrying a given PC; since A has no information about B, it has no way to determine whether the packet is freshly generated or a replay of a previously sent packet.

In this situation, A discards the packet and challenges B to prove that it knows the HMAC key. It sends a "challenge request", a TLV containing a unique nonce, a value that has never been used before and will never be used again. B replies to the challenge request with a "challenge reply", a TLV containing a copy of the nonce chosen by A, in a packet protected by HMAC and containing the new value of B's PC. Since the nonce has never been used before, B's reply proves B's knowledge of the HMAC key and the freshness of the PC.

By itself, this mechanism is safe against replay if B never resets its PC. In practice, however, this is difficult to ensure, as persistent storage is prone to failure, and hardware clocks, even when available, are occasionally reset. Suppose that B resets its PC to an earlier value, and sends a packet with a previously used PC  $n$ . A challenges B, B successfully responds to the challenge, and A accepts the PC equal to  $n + 1$ . At this point, an attacker C may send a replayed packet with PC equal to  $n + 2$ , which will be accepted by A.

Another mechanism is needed to protect against this attack. In this protocol, every PC is tagged with an "index", an arbitrary string of octets. Whenever B resets its PC, or whenever B doesn't know whether its PC has been reset, it picks an index that it has never used before (either by drawing it randomly or by using a reliable hardware clock) and starts sending PCs with that index. Whenever A detects that B has changed its index, it challenges B again.

With this additional mechanism, this protocol is invulnerable to replay attacks (see [Section 1.2](#) above).





### **3. Data Structures**

Every Babel node maintains a set of conceptual data structures described in Section 3.2 of [\[RFC6126bis\]](#). This protocol extends these data structures as follows.

#### **3.1. The Interface Table**

Every Babel node maintains an interface table, as described in [Section 3.2.3 \[RFC6126bis\]](#). Implementations of this protocol MUST allow each interface to be provisioned with a set of one or more HMAC keys and the associated HMAC algorithms (see [Section 4.1](#)). In order to allow incremental deployment of this protocol (see [Appendix A](#)), implementations SHOULD allow an interface to be configured in a mode in which it participates in the HMAC authentication protocol but accepts packets that are not authenticated.

This protocol extends each entry in this table that is associated with an interface on which HMAC authentication has been configured with two new pieces of data:

- o a set of one or more HMAC keys, each associated with a given HMAC algorithm ; the length of each key is exactly the hash size of the associated HMAC algorithm (i.e., the key is not subject to the preprocessing described in [Section 2 of \[RFC2104\]](#));
- o a pair (Index, PC), where Index is an arbitrary string of 0 to 32 octets, and PC is a 32-bit (4-octet) integer.

We say that an index is fresh when it has never been used before with any of the keys currently configured on the interface. The Index field is initialised to a fresh index, for example by drawing a random string of sufficient length, and the PC is initialised to an arbitrary value (typically 0).

#### **3.2. The Neighbour table**

Every Babel node maintains a neighbour table, as described in Section 3.2.4 of [\[RFC6126bis\]](#). This protocol extends each entry in this table with two new pieces of data:

- o a pair (Index, PC), where Index is a string of 0 to 32 octets, and PC is a 32-bit (4-octet) integer;
- o a Nonce, which is an arbitrary string of 0 to 192 octets, and an associated challenge expiry timer.



The Index and PC are initially undefined, and are managed as described in [Section 4.3](#). The Nonce and expiry timer are initially undefined, and used as described in [Section 4.3.1.1](#).

## 4. Protocol Operation

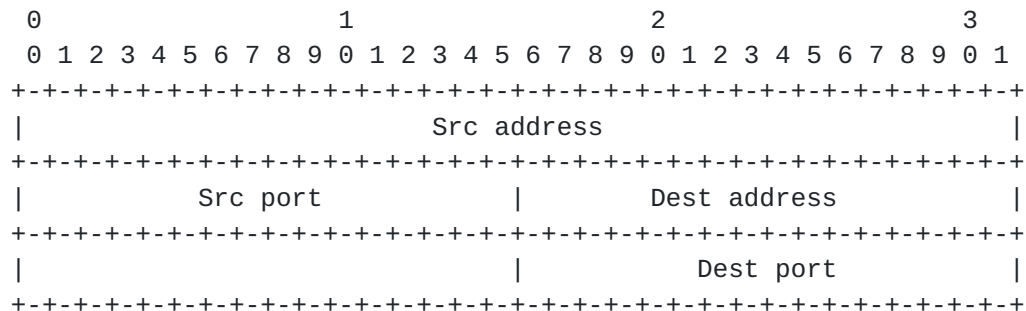
### 4.1. HMAC computation

A Babel node computes the HMAC of a Babel packet as follows.

First, the node builds a pseudo-header that will participate in HMAC computation but will not be sent. If the packet was carried over IPv6, the pseudo-header has the following format:



If the packet was carried over IPv4, the pseudo-header has the following format:





Fields :

Src address     The source IP address of the packet.

Src port        The source UDP port number of the packet.

Dest address    The destination IP address of the packet.

Src port        The destination UDP port number of the packet.

The node takes the concatenation of the pseudo-header and the packet including the packet header but excluding the packet trailer (from octet 0 inclusive up to (Body Length + 4) exclusive) and computes an HMAC with one of the implemented hash algorithms. Every implementation MUST implement HMAC-SHA256 as defined in [\[RFC6234\]](#) and [Section 2 of \[RFC2104\]](#), SHOULD implement keyed BLAKE2s [\[RFC7693\]](#), and MAY implement other HMAC algorithms.

#### **4.2. Packet Transmission**

A Babel node might delay actually sending TLVs by a small amount, in order to aggregate multiple TLVs in a single packet up to the interface MTU (Section 4 of [\[RFC6126bis\]](#)). For an interface on which HMAC protection is configured, the TLV aggregation logic MUST take into account the overhead due to PC TLVs (one in each packet) and HMAC TLVs (one per configured key).

Before sending a packet, the following actions are performed:

- o a PC TLV containing the PC and Index associated with the outgoing interface MUST be appended to the packet body; the PC MUST be incremented by a strictly positive amount (typically just 1); if the PC overflows, a fresh index MUST be generated (as defined in [Section 3.1](#)); a node MUST NOT include multiple PC TLVs in a single packet;
- o for each key configured on the interface, an HMAC is computed as specified in [Section 4.1](#) above, and stored in an HMAC TLV that MUST be appended to the packet trailer (see Section 4.2 of [\[RFC6126bis\]](#)).

#### **4.3. Packet Reception**

When a packet is received on an interface that is configured for HMAC protection, the following steps are performed before the packet is passed to normal processing:



- o First, the receiver checks whether the trailer of the received packet carries at least one HMAC TLV; if not, the packet MUST be immediately dropped and processing stops. Then, for each key configured on the receiving interface, the receiver computes the HMAC of the packet. It then compares every generated HMAC against every HMAC included in the packet; if there is at least one match, the packet passes the HMAC test; if there is none, the packet MUST be silently dropped and processing stops at this point. In order to avoid memory exhaustion attacks, an entry in the Neighbour Table MUST NOT be created before the HMAC test has passed successfully. The HMAC of the packet MUST NOT be computed for each HMAC TLV contained in the packet, but only once for each configured key.
- o The packet body is then parsed a first time. During this "preparse" phase, the packet body is traversed and all TLVs are ignored except PC TLVs, Challenge Requests and Challenge Replies. When a PC TLV is encountered, the enclosed PC and Index are saved for later processing; if multiple PCs are found (which should not happen, see [Section 4.2](#) above), only the first one is processed, the remaining ones MUST be silently ignored. If a Challenge Request is encountered, a Challenge Reply MUST be scheduled, as described in [Section 4.3.1.2](#). If a Challenge Reply is encountered, it is tested for validity as described in [Section 4.3.1.3](#) and a note is made of the result of the test.
- o The preparse phase above has yielded two pieces of data: the PC and Index from the first PC TLV, and a bit indicating whether the packet contains a successful Challenge Reply. If the packet does not contain a PC TLV, the packet MUST be dropped and processing stops at this point. If the packet contains a successful Challenge Reply, then the PC and Index contained in the PC TLV MUST be stored in the Neighbour Table entry corresponding to the sender (which may need to be created at this stage), and the packet is accepted.
- o Otherwise, if there is no entry in the Neighbour Table corresponding to the sender, or if such an entry exists but contains no Index, or if the Index it contains is different from the Index contained in the PC TLV, then a challenge MUST be sent as described in [Section 4.3.1.1](#), the packet MUST be dropped, and processing stops at this stage.
- o At this stage, the packet contains no successful challenge reply and the Index contained in the PC TLV is equal to the Index in the Neighbour Table entry corresponding to the sender. The receiver compares the received PC with the PC contained in the Neighbour Table; if the received PC is smaller or equal than the PC





contained in the Neighbour Table, the packet MUST be dropped and processing stops (no challenge is sent in this case, since the mismatch might be caused by harmless packet reordering on the link). Otherwise, the PC contained in the Neighbour Table entry is set to the received PC, and the packet is accepted.

In the algorithm described above, challenge requests are processed and challenges are sent before the PC/Index pair is verified against the neighbour table. This simplifies the implementation somewhat (the node may simply schedule outgoing requests as it walks the packet during the preparse phase), but relies on the rate-limiting described in [Section 4.3.1.1](#) to avoid sending too many challenges in response to replayed packets. As an optimisation, a node MAY ignore all challenge requests contained in a packet except the last one, and it MAY ignore a challenge request in the case where it is contained in a packet with an Index that matches the one in the Neighbour Table and a PC that is smaller or equal to the one contained in the Neighbour Table. Since it is still possible to replay a packet with an obsolete Index, the rate-limiting described in [Section 4.3.1.1](#) is required even if this optimisation is implemented.

The same is true of challenge replies. However, since validating a challenge reply is extremely cheap (it's just a bitwise comparison of two strings of octets), a similar optimisation for challenge replies is not worthwhile.

After the packet has been accepted, it is processed as normal, except that any PC, Challenge Request and Challenge Reply TLVs that it contains are silently ignored.

#### **[4.3.1.](#) Challenge Requests and Replies**

During the preparse stage, the receiver might encounter a mismatched Index, to which it will react by scheduling a Challenge Request. It might encounter a Challenge Request TLV, to which it will reply with a Challenge Reply TLV. Finally, it might encounter a Challenge Reply TLV, which it will attempt to match with a previously sent Challenge Request TLV in order to update the Neighbour Table entry corresponding to the sender of the packet.

##### **[4.3.1.1.](#) Sending challenges**

When it encounters a mismatched Index during the preparse phase, a node picks a nonce that it has never used with any of the keys currently configured on the relevant interface, for example by drawing a sufficiently large random string of bytes or by consulting a strictly monotonic hardware clock. It MUST then store the nonce in the entry of the Neighbour Table associated to the neighbour (the



entry might need to be created at this stage), initialise the neighbour's challenge expiry timer to 30 seconds, and send a Challenge Request TLV to the unicast address corresponding to the neighbour.

A node MAY aggregate a Challenge Request with other TLVs; in other words, if it has already buffered TLVs to be sent to the unicast address of the neighbour, it MAY send the buffered TLVs in the same packet as the Challenge Request. However, it MUST arrange for the Challenge Request to be sent in a timely manner, as any packets received from that neighbour will be silently ignored until the challenge completes.

Since a challenge may be prompted by a packet replayed by an attacker, a node MUST impose a rate limitation to the challenges it sends; the limit SHOULD default to one challenge request every 300ms, and MAY be configurable.

#### **4.3.1.2. Replying to challenges**

When it encounters a Challenge Request during the preparse phase, a node constructs a Challenge Reply TLV by copying the Nonce from the Challenge Request into the Challenge Reply. It MUST then send the Challenge Reply to the unicast address from which the Challenge Request was sent.

A node MAY aggregate a Challenge Reply with other TLVs; in other words, if it has already buffered TLVs to be sent to the unicast address of the sender of the Challenge Request, it MAY send the buffered TLVs in the same packet as the Challenge Reply. However, it MUST arrange for the Challenge Reply to be sent in a timely manner (within a few seconds), and SHOULD NOT send any other packets over the same interface before sending the Challenge Reply, as those would be dropped by the challenger.

A challenge sent to a multicast address MUST be silently ignored.

#### **4.3.1.3. Receiving challenge replies**

When it encounters a Challenge Reply during the preparse phase, a node consults the Neighbour Table entry corresponding to the neighbour that sent the Challenge Reply. If no challenge is in progress, i.e., if there is no Nonce stored in the Neighbour Table entry or the Challenge timer has expired, the Challenge Reply MUST be silently ignored and the challenge has failed.

Otherwise, the node compares the Nonce contained in the Challenge Reply with the Nonce contained in the Neighbour Table entry. If the



two are equal (they have the same length and content), then the challenge has succeeded; otherwise, the challenge has failed.

#### 4.4. Expiring per-neighbour state

The per-neighbour (Index, PC) pair is maintained in the neighbour table, and is normally discarded when the neighbour table entry expires. Implementations MUST ensure that an (Index, PC) pair is discarded within a finite time since the last time a packet has been accepted. In particular, unsuccessful challenges MUST NOT prevent an (Index, PC) pair from being discarded for unbounded periods of time.

A possible implementation strategy for implementations that use a Hello history (Appendix A of [\[RFC6126bis\]](#)) is to discard the (Index, PC) pair whenever the Hello history becomes empty. Another implementation strategy is to use a timer that is reset whenever a packet is accepted, and to discard the (Index, PC) pair whenever the timer expires. If the latter strategy is being used, the timer SHOULD default to a value of 5 min, and MAY be configurable.

### 5. Packet Format

#### 5.1. HMAC TLV

```

      0               1               2               3
      0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
|  Type = 16  |  Length  |  HMAC...  |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+

```

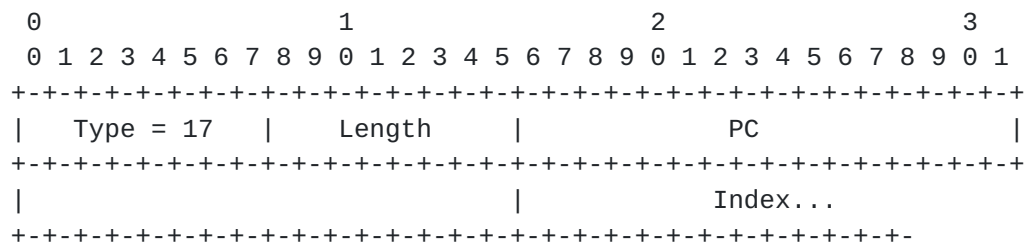
Fields :

Type	Set to 16 to indicate an HMAC TLV.
Length	The length of the body, in octets, exclusive of the Type and Length fields. The length of the body depends on the HMAC algorithm being used.
HMAC	The body contains the HMAC of the packet, computed as described in <a href="#">Section 4.1</a> .

This TLV is allowed in the packet trailer (see Section 4.2 of [\[RFC6126bis\]](#)), and MUST be ignored if it is found in the packet body.



## 5.2. PC TLV

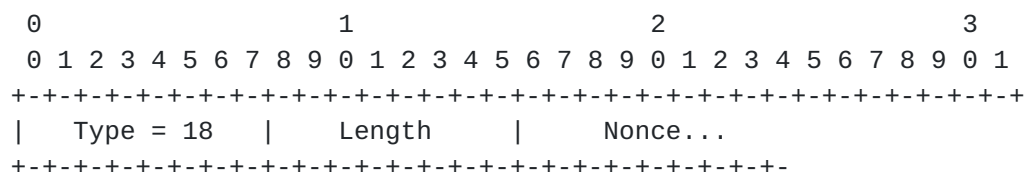


Fields :

- Type        Set to 17 to indicate a PC TLV.
- Length      The length of the body, in octets, exclusive of the Type and Length fields.
- PC          The Packet Counter (PC), a 32-bit (4 octet) unsigned integer which is increased with every packet sent over this interface. A fresh index (as defined in [Section 3.1](#)) MUST be generated whenever the PC overflows.
- Index       The sender's Index, an opaque string of 0 to 32 octets.

Indices are limited to a size of 32 octets: a node MUST NOT send a TLV with an index of size strictly larger than 32 octets, and a node MAY ignore a PC TLV with an index of length strictly larger than 32 octets. Indices of length 0 are valid: if a node has reliable stable storage and the packet counter never overflows, then only one index is necessary, and the value of length 0 is the canonical choice.

## 5.3. Challenge Request TLV



Fields :

- Type        Set to 18 to indicate a Challenge Request TLV.
- Length      The length of the body, in octets, exclusive of the Type and Length fields.
- Nonce       The nonce uniquely identifying the challenge, an opaque string of 0 to 192 octets.





Nonces are limited to a size of 192 octets: a node MUST NOT send a Challenge Request TLV with a nonce of size strictly larger than 192 octets, and a node MAY ignore a nonce that is of size strictly larger than 192 octets. Nonces of length 0 are valid: if a node has reliable stable storage, then it may use a sequential counter for generating nonces which get encoded in the minimum number of octets required; the value 0 is then encoded as the string of length 0.

#### 5.4. Challenge Reply TLV

```

      0                   1                   2                   3
      0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
|  Type = 19  |   Length   |      Nonce...      |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+

```

Fields :

Type	Set to 19 to indicate a Challenge Reply TLV.
Length	The length of the body, in octets, exclusive of the Type and Length fields.
Nonce	A copy of the nonce contained in the corresponding challenge request.

## 6. Security Considerations

This document defines a mechanism that provides basic security properties for the Babel routing protocol. The scope of this protocol is strictly limited: it only provides authentication (we assume that routing information is not confidential), it only supports symmetric keying, and it only allows for the use of a small number of symmetric keys on every link. Deployments that need more features, e.g., confidentiality or asymmetric keying, should use a more featureful security mechanism such as the one described in [\[I-D.ietf-babel-dtls\]](#).

This mechanism relies on two assumptions, as described in [Section 1.2](#). First, it assumes that the hash being used is invulnerable to pre-image attacks ([Section 1.1 of \[RFC6039\]](#)); at the time of writing, SHA-256, which is mandatory to implement ([Section 4.1](#)), is believed to be safe against practical attacks.

Second, it assumes that indices and nonces are generated uniquely over the lifetime of a key used for HMAC computation (more precisely, indices must be unique for a given (key, source) pair, and nonces must be unique for a given (key, source, destination) triple). This



property can be satisfied either by using a cryptographically secure random number generator to generate indices and nonces that contain enough entropy (64-bit values are believed to be large enough for all practical applications), or by using a reliably monotonic hardware clock. If uniqueness cannot be guaranteed (e.g., because a hardware clock has been reset), then rekeying is necessary.

The expiry mechanism mandated in [Section 4.4](#) is required to prevent an attacker from delaying an authentic packet by an unbounded amount of time. If an attacker is able to delay the delivery of a packet (e.g., because it is located at a layer 2 switch), then the packet will be accepted as long as the corresponding (Index, PC) pair is present at the receiver. If the attacker is able to cause the (Index, PC) pair to persist for arbitrary amounts of time (e.g., by repeatedly causing failed challenges), then it is able to delay the packet by arbitrary amounts of time, even after the sender has left the network.

While it is probably not possible to be immune against denial of service (DoS) attacks in general, this protocol includes a number of mechanisms designed to mitigate such attacks. In particular, reception of a packet with no correct HMAC creates no local state whatsoever ([Section 4.3](#)). Reception of a replayed packet with correct hash, on the other hand, causes a challenge to be sent; this is mitigated somewhat by requiring that challenges be rate-limited.

At first sight, sending a challenge requires retaining enough information to validate the challenge reply. However, the nonce included in a challenge request and echoed in the challenge reply can be fairly large (up to 192 octets), which should in principle permit encoding the per-challenge state as a secure "cookie" within the nonce itself.

## **7. IANA Considerations**

IANA has allocated the following values in the Babel TLV Types registry:



Type	Name	Reference
16	HMAC	this document
17	PC	this document
18	Challenge Request	this document
19	Challenge Reply	this document

## 8. Acknowledgments

The protocol described in this document is based on the original HMAC protocol defined by Denis Ovsienko [RFC7298]. The use of a pseudo-header was suggested by David Schinazi. The use of an index to avoid replay was suggested by Markus Stenberg. The authors are also indebted to Donald Eastlake, Toke Hoiland-Jorgensen, Florian Horn, Dave Taht and Martin Vigoureux.

## 9. References

### 9.1. Normative References

- [RFC2104] Krawczyk, H., Bellare, M., and R. Canetti, "HMAC: Keyed-Hashing for Message Authentication", [RFC 2104](#), DOI 10.17487/RFC2104, February 1997, <<https://www.rfc-editor.org/info/rfc2104>>.
- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", [BCP 14](#), [RFC 2119](#), DOI 10.17487/RFC2119, March 1997.
- [RFC6126bis] Chroboczek, J. and D. Schinazi, "The Babel Routing Protocol", [draft-ietf-babel-rfc6126bis-06](#) (work in progress), October 2018.
- [RFC6234] Eastlake 3rd, D. and T. Hansen, "US Secure Hash Algorithms (SHA and SHA-based HMAC and HKDF)", [RFC 6234](#), DOI 10.17487/RFC6234, May 2011, <<https://www.rfc-editor.org/info/rfc6234>>.
- [RFC7693] Saarinen, M-J., Ed. and J-P. Aumasson, "The BLAKE2 Cryptographic Hash and Message Authentication Code (MAC)", [RFC 7693](#), DOI 10.17487/RFC7693, November 2015, <<https://www.rfc-editor.org/info/rfc7693>>.



- [RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in [RFC 2119](#) Key Words", [BCP 14](#), [RFC 8174](#), DOI 10.17487/RFC8174, May 2017.

## 9.2. Informational References

- [I-D.ietf-babel-dtls]  
Decimo, A., Schinazi, D., and J. Chroboczek, "Babel Routing Protocol over Datagram Transport Layer Security", [draft-ietf-babel-dtls-01](#) (work in progress), October 2018.
- [RFC4086] Eastlake 3rd, D., Schiller, J., and S. Crocker, "Randomness Requirements for Security", [BCP 106](#), [RFC 4086](#), DOI 10.17487/RFC4086, June 2005, <<http://www.rfc-editor.org/info/rfc4086>>.
- [RFC6039] Manral, V., Bhatia, M., Jaeggli, J., and R. White, "Issues with Existing Cryptographic Protection Methods for Routing Protocols", [RFC 6039](#), DOI 10.17487/RFC6039, October 2010, <<https://www.rfc-editor.org/info/rfc6039>>.
- [RFC7298] Ovsienko, D., "Babel Hashed Message Authentication Code (HMAC) Cryptographic Authentication", [RFC 7298](#), DOI 10.17487/RFC7298, July 2014, <<https://www.rfc-editor.org/info/rfc7298>>.

## Appendix A. Incremental deployment and key rotation

This protocol supports incremental deployment (transitioning from an insecure network to a secured network with no service interruption) and key rotation (transitioning from a set of keys to a different set of keys).

In order to perform incremental deployment, the nodes in the network are first configured in a mode where packets are sent with authentication but not checked on reception. Once all the nodes in the network are configured to send authenticated packets, nodes are reconfigured to reject unauthenticated packets.

In order to perform key rotation, the new key is added to all the nodes; once this is done, both the old and the new key are sent in all packets, and packets are accepted if they are properly signed by either of the keys. At that point, the old key is removed.

In order to support incremental deployment and key rotation, implementations SHOULD support an interface configuration in which they send authenticated packets but accept all packets, and SHOULD





allow changing the set of keys associated with an interface without a restart.

## **Appendix B. Changes from previous versions**

[RFC Editor: please remove this section before publication.]

### **B.1. Changes since [draft-ietf-babel-hmac-00](#)**

- o Changed the title.
- o Removed the appendix about the packet trailer, this is now in rfc6126bis.
- o Removed the appendix with implicit indices.
- o Clarified the definitions of acronyms.
- o Limited the size of nonces and indices.

### **B.2. Changes since [draft-ietf-babel-hmac-01](#)**

- o Made BLAKE2s a recommended HMAC algorithm.
- o Added requirement to expire per-neighbour crypto state.

### **B.3. Changes since [draft-ietf-babel-hmac-02](#)**

- o Clarified that PCs are 32-bit unsigned integers.
- o Clarified that indices and nonces are of arbitrary size.
- o Added reference to [RFC 4086](#).

### **B.4. Changes since [draft-ietf-babel-hmac-03](#)**

- o Use the TLV values allocated by IANA.
- o Fixed an issue with packets that contain a successful challenge reply: they should be accepted before checking the PC value.
- o Clarified that keys are the exact value of the HMAC hash size, and not subject to preprocessing; this makes management more deterministic.



**B.5. Changes since [draft-ietf-babel-hmac-04](#)**

- o Use normative language in more places.

**B.6. Changes since [draft-ietf-babel-hmac-05](#)**

- o Do not update RFC 6126bis.
- o Clarify that indices and nonces of length 0 are valid.
- o Clarify that multiple PC TLVs in a single packet are not allowed.
- o Allow discarding challenge requests when they carry an old PC.

**Authors' Addresses**

Clara Do  
IRIF, University of Paris-Diderot  
75205 Paris Cedex 13  
France

Email: [clarado\\_perso@yahoo.fr](mailto:clarado_perso@yahoo.fr)

Weronika Kolodziejak  
IRIF, University of Paris-Diderot  
75205 Paris Cedex 13  
France

Email: [weronika.kolodziejak@gmail.com](mailto:weronika.kolodziejak@gmail.com)

Juliusz Chroboczek  
IRIF, University of Paris-Diderot  
Case 7014  
75205 Paris Cedex 13  
France

Email: [jch@irif.fr](mailto:jch@irif.fr)

