Internet Engineering Task Force (IETF)
Request for Comments: 8967
Obsoletes: 7298
Category: Standards Track
ISSN: 2070-1721

C. Dô
W. Kolodziejak
J. Chroboczek
IRIF, University of Paris-Diderot

January 2021

MAC Authentication for the Babel Routing Protocol

Abstract

This document describes a cryptographic authentication mechanism for the Babel routing protocol that has provisions for replay avoidance. This document obsoletes RFC 7298.

Status of This Memo

This is an Internet Standards Track document.

This document is a product of the Internet Engineering Task Force (IETF). It represents the consensus of the IETF community. It has received public review and has been approved for publication by the Internet Engineering Steering Group (IESG). Further information on Internet Standards is available in Section 2 of RFC 7841.

Information about the current status of this document, any errata, and how to provide feedback on it may be obtained at https://www.rfc-editor.org/info/rfc8967.

Table of Contents

1. Introduction

By default, the Babel routing protocol [RFC8966] 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:

* spoof a Babel packet and redirect traffic by announcing a route with a smaller metric, a larger sequence number, or a longer prefix;

* spoof a malformed packet, which could cause an insufficiently robust implementation to crash or interfere with the rest of the network;

* 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 [RFC8968], 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. This document obsoletes [RFC7298].

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:

* that the Message Authentication Code (MAC) being used is invulnerable to forgery, i.e., that an attacker is unable to generate a packet with a correct MAC without access to the secret key;

* 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 MAC 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 often enough that collisions are unlikely.

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

* it is invulnerable to spoofing: any Babel packet accepted as authentic is the exact copy of a packet originally sent by an authorised node;

* 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;

* among different nodes, it is only vulnerable to immediate replay: if a node A has accepted an authentic packet from C, then a node B will only accept a copy of that packet if B has accepted an older packet from C, and B has received no later packet from C.

While this protocol makes 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 MAC cryptographic protection, it computes one or more MACs (one per key) that it appends to the packet. When a node A receives a packet over an interface that requires MAC cryptographic protection, it independently computes a set of MACs and compares them to the MACs 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 MACs 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, peer A discards the packet and challenges B to prove that it knows the MAC 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. Peer 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 MAC 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 MAC 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. Peer 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).

3. Data Structures

Every Babel node maintains a set of conceptual data structures described in Section 3.2 of [RFC8966]. 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 of [RFC8966]. Implementations of this protocol MUST allow each interface to be provisioned with a set of one or more MAC keys and the associated MAC algorithms (see Section 4.1 for suggested algorithms and Section 7 for suggested methods for key generation). In order to allow incremental deployment of this protocol (see Section 5), implementations SHOULD allow an interface to be configured in a mode in which it participates in the MAC authentication protocol but accepts packets that are not authenticated.

This protocol extends each table entry associated with an interface on which MAC authentication has been configured with two new pieces of data:

* a set of one or more MAC keys, each associated with a given MAC algorithm;

* 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 (see Section 7 for suggested sizes), 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 [RFC8966]. This protocol extends each entry in this table with two new pieces of data:

* a pair (Index, PC), where Index is a string of 0 to 32 octets, and PC is a 32-bit (4-octet) integer;

* a Nonce, which is an arbitrary string of 0 to 192 octets, and an associated challenge expiry timer.

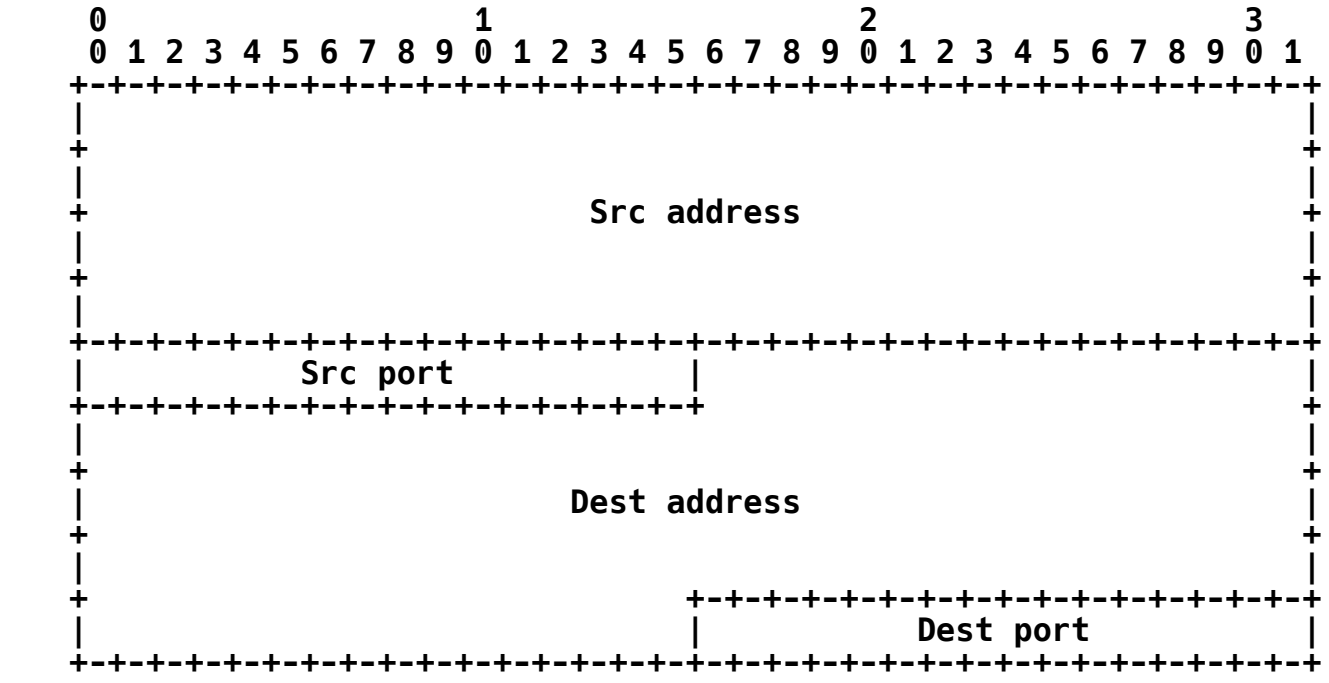
The Index and PC are initially undefined, and they are managed as described in Section 4.3. The Nonce and challenge expiry timer are initially undefined, and they are used as described in Section 4.3.1.1.

4. Protocol Operation

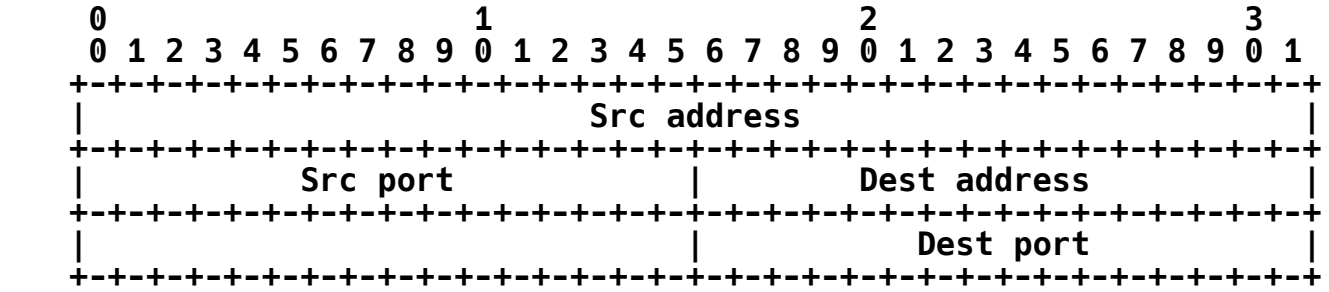
4.1. MAC Computation

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

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



If the packet is 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 Babel packet including the packet header but excluding the packet trailer (from octet 0 inclusive up to (Body Length + 4) exclusive) and

computes a MAC with one of the implemented algorithms. Every implementation MUST implement HMAC-SHA256 as defined in [RFC6234] and Section 2 of [RFC2104], SHOULD implement keyed BLAKE2s [RFC7693] with 128-bit (16-octet) digests, and MAY implement other MAC 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 [RFC8966]). For an interface on which MAC protection is configured, the TLV aggregation logic MUST take into account the overhead due to PC TLVs (one in each packet) and MAC TLVs (one per configured key).

Before sending a packet, the following actions are performed:

* 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;

* for each key configured on the interface, a MAC is computed as specified in Section 4.1 and stored in a MAC TLV that MUST be appended to the packet trailer (see Section 4.2 of [RFC8966]).

4.3. Packet Reception

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

* First, the receiver checks whether the trailer of the received packet carries at least one MAC 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 MAC of the packet. It then compares every generated MAC against every MAC included in the packet; if there is at least one match, the packet passes the MAC 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 MAC test has passed successfully. The MAC of the packet MUST NOT be computed for each MAC TLV contained in the packet, but only once for each configured key.

* If an entry for the sender does not exist in the neighbour table, it MAY be created at this point (or, alternatively, its creation can be delayed until a challenge needs to be sent, see below).

* 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, Challenge Request, and Challenge Reply TLVs. 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), 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.

* The preparse phase above yields 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 already exists in this case), and the packet is accepted.

* 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.

* 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 (Index, PC) 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 has minimal additional cost (it is 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.

A node MUST impose a rate limitation to the challenges it sends; the limit SHOULD default to one Challenge Request every 300 ms and MAY be configurable. This rate limiting serves two purposes. First, since a challenge may be sent in response to a packet replayed by an attacker, it limits the number of challenges that an attacker can cause a node to send. Second, it limits the number of challenges sent when there are multiple packets in flight from a single neighbour.

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 challenge sent to a multicast address MUST be silently ignored.

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.

Since a Challenge Reply might be caused by a replayed Challenge Request, a node MUST impose a rate limitation to the Challenge Replies it sends; the limit SHOULD default to one Challenge Reply for each peer every 300 ms and MAY be configurable.

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 and the nonce stored in the neighbour table for this neighbour SHOULD be discarded; otherwise, the challenge has failed (and the nonce is not discarded).

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 [RFC8966]) 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 used, the timer SHOULD default to a value of 5 minutes and MAY be configurable.

5. Incremental Deployment and Key Rotation

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.

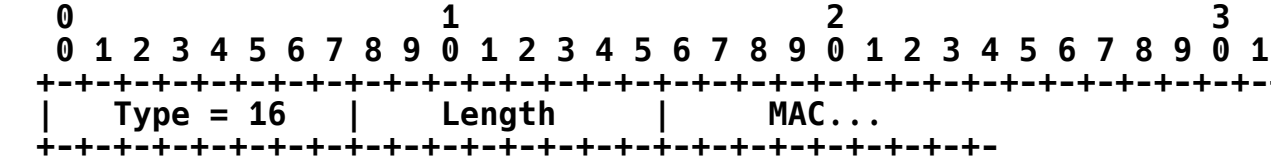
In order to perform key rotation, the new key is added to all the

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 the procedures described above, implementations of this protocol SHOULD support an interface configuration in which packets are sent authenticated but received packets are accepted without verification, and they SHOULD allow changing the set of keys associated with an interface without a restart.

6. Packet Format

6.1. MAC TLV



Fields:

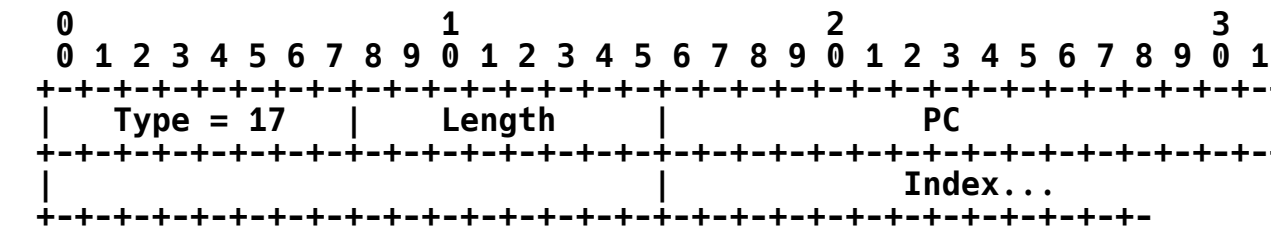
Type — Set to 16 to indicate a MAC TLV.

Length — The length of the body, in octets, exclusive of the Type and Length fields. The length depends on the MAC algorithm being used.

MAC — The body contains the MAC of the packet, computed as described in Section 4.1.

This TLV is allowed in the packet trailer (see Section 4.2 of [RFC8966]) and MUST be ignored if it is found in the packet body.

6.2. PC TLV



Fields:

Type — Set to 17 to indicate a PC TLV.

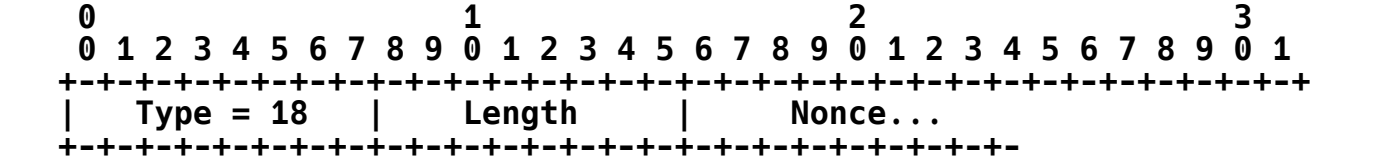
Length — The length of the body, in octets, exclusive of the Type and Length fields.

PC — The Packet Counter (PC), a 32-bit (4-octet) unsigned integer that 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.

6.3. Challenge Request TLV



Fields:

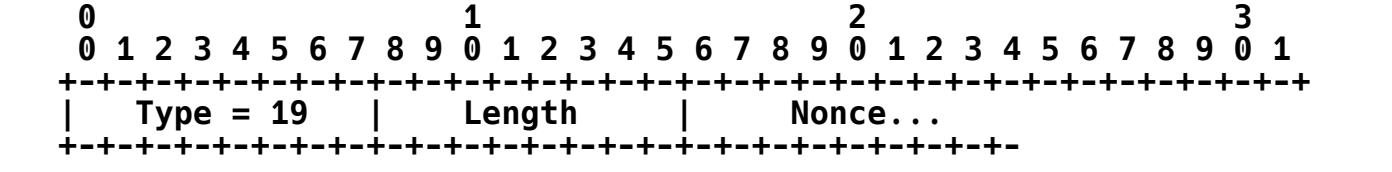
Type Set to 18 to indicate a Challenge Request TLV.

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 that get encoded in the minimum number of octets required; the value 0 is then encoded as the string of length 0.

6.4. Challenge Reply TLV



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.

7. 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 feature-rich security mechanism such as the one described in [RFC8968].

This mechanism relies on two assumptions, as described in Section 1.2. First, it assumes that the MAC being used is invulnerable to forgery (Section 1.1 of [RFC6039]); at the time of writing, HMAC-SHA256, 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 MAC 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, which could allow it to redirect or blackhole traffic to destinations previously advertised by the sender.

This protocol exposes large numbers of packets and their MACs to an attacker that is able to capture packets; it is therefore vulnerable to brute-force attacks. Keys must be chosen in a manner that makes them difficult to guess. Ideally, they should have a length of 32 octets (both for HMAC-SHA256 and BLAKE2s), and be chosen randomly. If, for some reason, it is necessary to derive keys from a human-readable passphrase, it is recommended to use a key derivation function that hampers dictionary attacks, such as PBKDF2 [RFC8018], bcrypt [BCRYPT], or scrypt [RFC7914]. In that case, only the derived keys should be communicated to the routers; the original passphrase itself should be kept on the host used to perform the key generation (e.g., an administrator's secure laptop computer).

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 MAC creates no local Babel state (Section 4.3). Reception of a replayed packet with correct

MAC, on the other hand, causes a challenge to be sent; this is mitigated somewhat by requiring that challenges be rate limited (Section 4.3.1.1).

Receiving a replayed packet with an obsolete index causes an entry to be created in the neighbour table, which, at first sight, makes the protocol susceptible to resource exhaustion attacks (similarly to the familiar "TCP SYN Flooding" attack [RFC4987]). However, the MAC computation includes the sender address (Section 4.1), and thus the amount of storage that an attacker can force a node to consume is limited by the number of distinct source addresses used with a single MAC key (see also Section 4 of [RFC8966], which mandates that the source address is a link-local IPv6 address or a local IPv4 address).

In order to make this kind of resource exhaustion attacks less effective, implementations may use a separate table of uncompleted challenges that is separate from the neighbour table used by the core protocol (the data structures described in Section 3.2 of [RFC8966] are conceptual, and any data structure that yields the same result may be used). Implementers might also consider using the fact that the nonces included in Challenge Requests and Replies can be fairly large (up to 192 octets), which should in principle allow encoding the per-challenge state as a secure "cookie" within the nonce itself; note, however, that any such scheme will need to prevent cookie replay.

8. IANA Considerations

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

Type	Name	Reference
16	MAC	RFC 8967
17	PC	RFC 8967
18	Challenge Request	RFC 8967
19	Challenge Reply	RFC 8967

Table 1

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,
<https://www.rfc-editor.org/info/rfc2119>.

[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, <https://www.rfc-editor.org/info/rfc8174>.

[RFC8966] Chroboczek, J. and D. Schinazi, "The Babel Routing Protocol", RFC 8966, DOI 10.17487/RFC8966, January 2021, <https://www.rfc-editor.org/info/rfc8966>.

9.2. Informational References

[BCRYPT] Niels, P. and D. Mazières, "A Future-Adaptable Password Scheme", Proceedings of the FREENIX Track: 1999 USENIX Annual Technical Conference, June 1999.

[RFC4086] Eastlake 3rd, D., Schiller, J., and S. Crocker, "Randomness Requirements for Security", BCP 106, RFC 4086, DOI 10.17487/RFC4086, June 2005, <https://www.rfc-editor.org/info/rfc4086>.

[RFC4987] Eddy, W., "TCP SYN Flooding Attacks and Common Mitigations", RFC 4987, DOI 10.17487/RFC4987, August 2007, <https://www.rfc-editor.org/info/rfc4987>.

[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>.

[RFC7914] Percival, C. and S. Josefsson, "The scrypt Password-Based Key Derivation Function", RFC 7914, DOI 10.17487/RFC7914, August 2016, <https://www.rfc-editor.org/info/rfc7914>.

[RFC8018] Moriarty, K., Ed., Kaliski, B., and A. Rusch, "PKCS #5: Password-Based Cryptography Specification Version 2.1", RFC 8018, DOI 10.17487/RFC8018, January 2017, <https://www.rfc-editor.org/info/rfc8018>.

[RFC8968] Décimo, A., Schinazi, D., and J. Chroboczek, "Babel

Routing Protocol over Datagram Transport Layer Security",
RFC 8968, DOI 10.17487/RFC8968, January 2021,
<https://www.rfc-editor.org/info/rfc8968>.

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 Antonin Décimo, Donald Eastlake, Toke Høiland-Jørgensen, Florian Horn, Benjamin Kaduk, Dave Taht, and Martin Vigoureux.

Authors' Addresses

Clara Dô
IRIF, University of Paris-Diderot
75205 Paris CEDEX 13
France

Email: clarado_perso@yahoo.fr

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

Email: weronika.kolodziejak@gmail.com

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

Email: jch@irif.fr