QUIC
Internet-Draft
Intended status: Standards Track
Expires: 8 August 2024

M. Duke
Google
N. Banks
Microsoft
C. Huitema
Private Octopus Inc.

5 February 2024

QUIC-LB: Generating Routable QUIC Connection IDs
draft-ietf-quic-load-balancers-19

Abstract

QUIC address migration allows clients to change their IP address while maintaining connection state. To reduce the ability of an observer to link two IP addresses, clients and servers use new connection IDs when they communicate via different client addresses. This poses a problem for traditional "layer-4" load balancers that route packets via the IP address and port 4-tuple. This specification provides a standardized means of securely encoding routing information in the server's connection IDs so that a properly configured load balancer can route packets with migrated addresses correctly. As it proposes a structured connection ID format, it also provides a means of connection IDs self-encoding their length to aid some hardware offloads.

Table of Contents

1. Introduction 4
  1.1. Terminology 5
  1.2. Notation 5
2. First CID octet 5
  2.1. Config Rotation 5
  2.2. Configuration Failover 6
  2.3. Length Self-Description 7
  2.4. Format 7
3. Load Balancing Preliminaries 8
  3.1. Unroutable Connection IDs 8
  3.2. Fallback Algorithms 9
  3.3. Server ID Allocation 10
4. Server ID Encoding in Connection IDs 10
  4.1. CID format 10
  4.2. Configuration Agent Actions 11
  4.3. Server Actions 11
    4.3.1. Special Case: Single Pass Encryption 12
    4.3.2. General Case: Four-Pass Encryption 12
  4.4. Load Balancer Actions 17
    4.4.1. Special Case: Single Pass Encryption 17
    4.4.2. General Case: Four-Pass Encryption 17
5. Per-connection state 18
6. Additional Use Cases 19
  6.1. Load balancer chains 19
  6.2. Server Process Demultiplexing 19
  6.3. Moving connections between servers 20
7. Version Invariance of QUIC-LB 20
8. Security Considerations 22
  8.1. Attackers not between the load balancer and server . . . 22
  8.2. Attackers between the load balancer and server 23
  8.3. Multiple Configuration IDs 23
  8.4. Limited configuration scope 23
  8.5. Stateless Reset Oracle 24
  8.6. Connection ID Entropy 24
  8.7. Distinguishing Attacks 25
9. IANA Considerations 26
10. References 26

10.1. Normative References 26
10.2. Informative References 26
Appendix A. QUIC-LB YANG Model 28
A.1. Tree Diagram 34
Appendix B. Load Balancer Test Vectors 34
B.1. Unencrypted CIDs 34
B.2. Encrypted CIDs 34
Appendix C. Interoperability with DTLS over UDP 35
C.1. DTLS 1.0 and 1.2 35
C.2. DTLS 1.3 36
C.3. Future Versions of DTLS 37
Appendix D. Acknowledgments 37
Appendix E. Change Log 37
E.1. since draft-ietf-quic-load-balancers-18 37
E.2. since draft-ietf-quic-load-balancers-17 37
E.3. since draft-ietf-quic-load-balancers-16 37
E.4. since draft-ietf-quic-load-balancers-15 37
E.5. since draft-ietf-quic-load-balancers-14 37
E.6. since draft-ietf-quic-load-balancers-13 38
E.7. since draft-ietf-quic-load-balancers-12 38
E.8. since draft-ietf-quic-load-balancers-11 38
E.9. since draft-ietf-quic-load-balancers-10 38
E.10. since draft-ietf-quic-load-balancers-09 38
E.11. since draft-ietf-quic-load-balancers-08 38
E.12. since draft-ietf-quic-load-balancers-07 39
E.13. since draft-ietf-quic-load-balancers-06 39
E.14. since draft-ietf-quic-load-balancers-05 39
E.15. since draft-ietf-quic-load-balancers-04 39
E.16. since-draft-ietf-quic-load-balancers-03 40
E.17. since-draft-ietf-quic-load-balancers-02 40
E.18. since-draft-ietf-quic-load-balancers-01 40
E.19. since-draft-ietf-quic-load-balancers-00 40
E.20. Since draft-duke-quic-load-balancers-06 40
E.21. Since draft-duke-quic-load-balancers-05 40
E.22. Since draft-duke-quic-load-balancers-04 41
E.23. Since draft-duke-quic-load-balancers-03 41
E.24. Since draft-duke-quic-load-balancers-02 41
E.25. Since draft-duke-quic-load-balancers-01 41
E.26. Since draft-duke-quic-load-balancers-00 41
Authors' Addresses 41

1. Introduction

QUIC packets [RFC9000] usually contain a connection ID to allow endpoints to associate packets with different address/port 4-tuples to the same connection context. This feature makes connections robust in the event of NAT rebinding. QUIC endpoints usually designate the connection ID which peers use to address packets. Server-generated connection IDs create a potential need for out-of-band communication to support QUIC.

QUIC allows servers (or load balancers) to encode useful routing information for load balancers in connection IDs. It also encourages servers, in packets protected by cryptography, to provide additional connection IDs to the client. This allows clients that know they are going to change IP address or port to use a separate connection ID on the new path, thus reducing linkability as clients move through the world.

There is a tension between the requirements to provide routing information and mitigate linkability. Ultimately, because new connection IDs are in protected packets, they must be generated at the server if the load balancer does not have access to the connection keys. However, it is the load balancer that has the context necessary to generate a connection ID that encodes useful routing information. In the absence of any shared state between load balancer and server, the load balancer must maintain a relatively expensive table of server-generated connection IDs, and will not route packets correctly if they use a connection ID that was originally communicated in a protected NEW_CONNECTION_ID frame.

This specification provides common algorithms for encoding the server mapping in a connection ID given some shared parameters. The mapping is generally only discoverable by observers that have the parameters, preserving unlinkability as much as possible.

As this document proposes a structured QUIC Connection ID, it also proposes a system for self-encoding connection ID length in all packets, so that crypto offload can efficiently obtain key information.

While this document describes a small set of configuration parameters to make the server mapping intelligible, the means of distributing these parameters between load balancers, servers, and other trusted intermediaries is out of its scope. There are numerous well-known infrastructures for distribution of configuration.

1.1. Terminology

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 [RFC2119].

In this document, these words will appear with that interpretation only when in ALL CAPS. Lower case uses of these words are not to be interpreted as carrying significance described in RFC 2119.

In this document, "client" and "server" refer to the endpoints of a QUIC connection unless otherwise indicated. A "load balancer" is an intermediary for that connection that does not possess QUIC connection keys, but it may rewrite IP addresses or conduct other IP or UDP processing. A "configuration agent" is the entity that determines the QUIC-LB configuration parameters for the network and leverages some system to distribute that configuration.

Note that stateful load balancers that act as proxies, by terminating a QUIC connection with the client and then retrieving data from the server using QUIC or another protocol, are treated as a server with respect to this specification.

For brevity, "Connection ID" will often be abbreviated as "CID".

1.2. Notation

All wire formats will be depicted using the notation defined in Section 1.3 of [RFC9000].

2. First CID octet

The Connection ID construction schemes defined in this document reserve the first octet of a CID for two special purposes: one mandatory (config rotation) and one optional (length self-description).

Subsequent sections of this document refer to the contents of this octet as the "first octet."

2.1. Config Rotation

The first three bits of any connection ID MUST encode an identifier for the configuration that the connection ID uses. This enables incremental deployment of new QUIC-LB settings (e.g., keys).

When new configuration is distributed to servers, there will be a transition period when connection IDs reflecting old and new configuration coexist in the network. The rotation bits allow load balancers to apply the correct routing algorithm and parameters to incoming packets.

Configuration Agents SHOULD deliver new configurations to load balancers before doing so to servers, so that load balancers are ready to process CIDs using the new parameters when they arrive.

A Configuration Agent SHOULD NOT use a codepoint to represent a new configuration until it takes precautions to make sure that all connections using CIDs with an old configuration at that codepoint have closed or transitioned.

Servers MUST NOT generate new connection IDs using an old configuration after receiving a new one from the configuration agent. Servers MUST send NEW_CONNECTION_ID frames that provide CIDs using the new configuration, and retire CIDs using the old configuration using the "Retire Prior To" field of that frame.

It also possible to use these bits for more long-lived distinction of different configurations, but this has privacy implications (see Section 8.3).

2.2. Configuration Failover

A server that is configured to use QUIC-LB might be forced to accept new connections without having received a current configuration. A server without QUIC-LB configuration can accept connections, but it SHOULD generate initial connection IDs with the config rotation bits set to 0b111 and avoid sending the client connection IDs in NEW_CONNECTION_ID frames or the preferred_address transport parameter. Servers in this state SHOULD use the "disable_active_migration" transport parameter until a valid configuration is received.

A load balancer that sees a connection ID with config rotation bits set to 0b111 MUST route using an algorithm based solely on the address/port 4-tuple, which is consistent well beyond the QUIC handshake. However, a load balancer MAY observe the connection IDs used during the handshake and populate a connection ID table that allows the connection to survive a NAT rebinding, and reduces the probability of connection failure due to a change in the number of servers.

When using codepoint 0b111, all bytes but the first SHOULD have no larger of a chance of collision as random bytes. The connection ID SHOULD be of at least length 8 to provide 7 bytes of entropy after the first octet with a low chance of collision. Furthermore, servers in a pool SHOULD also use a consistent connection ID length to simplify the load balancer's extraction of a connection ID from short headers.

2.3. Length Self-Description

Local hardware cryptographic offload devices may accelerate QUIC servers by receiving keys from the QUIC implementation indexed to the connection ID. However, on physical devices operating multiple QUIC servers, it might be impractical to efficiently lookup keys if the connection ID varies in length and does not self-encode its own length.

Note that this is a function of particular server devices and is irrelevant to load balancers. As such, load balancers MAY omit this from their configuration. However, the remaining 5 bits in the first octet of the Connection ID are reserved to express the length of the following connection ID, not including the first octet.

A server not using this functionality SHOULD choose the five bits so as to have no observable relationship to previous connection IDs issued for that connection.

2.4. Format

```
First Octet {
  Config Rotation (3),
  CID Len or Random Bits (5),
}
```

Figure 1: First Octet Format

The first octet has the following fields:

Config Rotation: Indicates the configuration used to interpret the CID.

CID Len or Random Bits: Length Self-Description (if applicable), or random bits otherwise. Encodes the length of the Connection ID following the First Octet.

3. Load Balancing Preliminaries

In QUIC-LB, load balancers do not generate individual connection IDs for servers. Instead, they communicate the parameters of an algorithm to generate routable connection IDs.

The algorithms differ in the complexity of configuration at both load balancer and server. Increasing complexity improves obfuscation of the server mapping.

This section describes three participants: the configuration agent, the load balancer, and the server. For any given QUIC-LB configuration that enables connection-ID-aware load balancing, there must be a choice of (1) routing algorithm, (2) server ID allocation strategy, and (3) algorithm parameters.

Fundamentally, servers generate connection IDs that encode their server ID. Load balancers decode the server ID from the CID in incoming packets to route to the correct server.

There are situations where a server pool might be operating two or more routing algorithms or parameter sets simultaneously. The load balancer uses the first two bits of the connection ID to multiplex incoming DCIDs over these schemes (see Section 2.1).

3.1. Unroutable Connection IDs

QUIC-LB servers will generate Connection IDs that are decodable to extract a server ID in accordance with a specified algorithm and parameters. However, QUIC often uses client-generated Connection IDs prior to receiving a packet from the server.

These client-generated CIDs might not conform to the expectations of the routing algorithm and therefore not be routable by the load balancer. Those that are not routable are "unroutable DCIDs" and receive similar treatment regardless of why they’re unroutable:

* The config rotation bits (Section 2.1) may not correspond to an active configuration. Note: a packet with a DCID with config ID codepoint 0b111 (see Section 2.2) is always routable.

* The DCID might not be long enough for the decoder to process.

* The extracted server mapping might not correspond to an active server.

All other DCIDs are routable.

Load balancers MUST forward packets with routable DCIDs to a server in accordance with the chosen routing algorithm. Exception: if the load balancer can parse the QUIC packet and makes a routing decision depending on the contents (e.g., the SNI in a TLS client hello), it MAY route in accordance with this instead. However, load balancers MUST always route long header packets it cannot parse in accordance with the DCID (see Section 7).

Load balancers SHOULD drop short header packets with unroutable DCIDs.

When forwarding a packet with a long header and unroutable DCID, load balancers MUST use a fallback algorithm as specified in Section 3.2.

Load balancers MAY drop packets with long headers and unroutable DCIDs if and only if it knows that the encoded QUIC version does not allow an unroutable DCID in a packet with that signature. For example, a load balancer can safely drop a QUIC version 1 Handshake packet with an unroutable DCID, as a version 1 Handshake packet sent to a QUIC-LB routable server will always have a server-generated routable CID. The prohibition against dropping packets with long headers remains for unknown QUIC versions.

Furthermore, while the load balancer function MUST NOT drop packets, the device might implement other security policies, outside the scope of this specification, that might force a drop.

Servers that receive packets with unroutable CIDs MUST use the available mechanisms to induce the client to use a routable CID in future packets. In QUIC version 1, this requires using a routable CID in the Source CID field of server-generated long headers.

3.2. Fallback Algorithms

There are conditions described below where a load balancer routes a packet using a "fallback algorithm." It can choose any algorithm, without coordination with the servers, but the algorithm SHOULD be deterministic over short time scales so that related packets go to the same server. The design of this algorithm SHOULD consider the version-invariant properties of QUIC described in [RFC8999] to maximize its robustness to future versions of QUIC.

A fallback algorithm MUST NOT make the routing behavior dependent on any bits in the first octet of the QUIC packet header, except the first bit, which indicates a long header. All other bits are QUIC version-dependent and intermediaries SHOULD NOT base their design on version-specific templates.

For example, one fallback algorithm might convert a unroutable DCID to an integer and divided by the number of servers, with the modulus used to forward the packet. The number of servers is usually consistent on the time scale of a QUIC connection handshake. Another might simply hash the address/port 4-tuple. See also Section 7.

3.3. Server ID Allocation

Load Balancer configurations include a mapping of server IDs to forwarding addresses. The corresponding server configurations contain one or more unique server IDs.

The configuration agent chooses a server ID length for each configuration that MUST be at least one octet.

A QUIC-LB configuration MAY significantly over-provision the server ID space (i.e., provide far more codepoints than there are servers) to increase the probability that a randomly generated Destination Connection ID is unroutable.

The configuration agent SHOULD provide a means for servers to express the number of server IDs it can usefully employ, because a single routing address actually corresponds to multiple server entities (see Section 6.1).

Conceptually, each configuration has its own set of server ID allocations, though two static configurations with identical server ID lengths MAY use a common allocation between them.

A server encodes one of its assigned server IDs in any CID it generates using the relevant configuration.

4. Server ID Encoding in Connection IDs

4.1. CID format

All connection IDs use the following format:

```
QUIC-LB Connection ID {
    First Octet (8),
    Plaintext Block (40..152),
}
Plaintext Block {
    Server ID (8..),
    Nonce (32..),
}
```

Figure 2: CID Format

The First Octet field serves one or two purposes, as defined in Section 2.

The Server ID field encodes the information necessary for the load balancer to route a packet with that connection ID. It is often encrypted.

The server uses the Nonce field to make sure that each connection ID it generates is unique, even though they all use the same Server ID.

4.2. Configuration Agent Actions

The configuration agent assigns a server ID to every server in its pool in accordance with Section 3.3, and determines a server ID length (in octets) sufficiently large to encode all server IDs, including potential future servers.

Each configuration specifies the length of the Server ID and Nonce fields, with limits defined for each algorithm.

Optionally, it also defines a 16-octet key. Note that failure to define a key means that observers can determine the assigned server of any connection, significantly increasing the linkability of QUIC address migration.

The nonce length MUST be at least 4 octets. The server ID length MUST be at least 1 octet.

As QUIC version 1 limits connection IDs to 20 octets, the server ID and nonce lengths MUST sum to 19 octets or less.

4.3. Server Actions

The server writes the first octet and its server ID into their respective fields.

If there is no key in the configuration, the server MUST fill the Nonce field with bytes that have no observable relationship to the field in previously issued connection IDs. If there is a key, the server fills the nonce field with a nonce of its choosing. See Section 8.6 for details.

The server MAY append additional bytes to the connection ID, up to the limit specified in that version of QUIC, for its own use. These bytes MUST NOT provide observers with any information that could link two connection IDs to the same connection, client, or server. In particular, all servers using a configuration MUST consistently add the same length to each connection ID, to preserve the linkability objectives of QUIC-LB. Any additional bytes SHOULD NOT provide any observable correlation to previous connection IDs for that connection (e.g., the bytes can be chosen at random).

If there is no key in the configuration, the Connection ID is complete. Otherwise, there are further steps, as described in the two following subsections.

Encryption below uses the AES-128-ECB cipher [NIST-AES-ECB]. Future standards could add new algorithms that use other ciphers to provide cryptographic agility in accordance with [RFC7696]. QUIC-LB implementations SHOULD be extensible to support new algorithms.

4.3.1. Special Case: Single Pass Encryption

When the nonce length and server ID length sum to exactly 16 octets, the server MUST use a single-pass encryption algorithm. All connection ID octets except the first form an AES-ECB block. This block is encrypted once, and the result forms the second through seventeenth most significant bytes of the connection ID.

4.3.2. General Case: Four-Pass Encryption

Any other field length requires four passes for encryption and at least three for decryption. To understand this algorithm, it is useful to define four functions that minimize the amount of bit-shifting necessary in the event that there are an odd number of octets.

When configured with both a key, and a nonce length and server ID length that sum to any number other than 16, the server MUST follow the algorith below to encrypt the connection ID.

4.3.2.1. Overview

The 4-pass algorithm is a four-round Feistel Network with the round function being AES-ECB. Most modern applications of Feistel Networks have more than four rounds. The implications of this choice, which is meant to limit the per-packet compute overhead at load balancers, are discussed in Section 8.7.

The server concatenates the server ID and nonce into a single field, which is then split into equal halves. In successive passes, one of these halves is expanded into a 16B plaintext, encrypted with AES-ECB, and the result XORed with the other half. The diagram below shows the conceptual processing of a plaintext server ID and nonce into a connection ID. 'FO' stands for 'First Octet'.

```
      +-----+-----------+-----------------------+
      | FO  | Server ID |         Nonce         |
      +--+--+-----------+-----+-----------------+
         |                    |
         |                    V
         |  +-----------------+-----------------+
         |  |      left_0     |     right_0     |
         |  +--+--------------+--------------+--+
         |     |                             |
         |     |                             |
         |     |         .--------.          V
         |     +-------->| AES-ECB +-------->
         |     |         '________'          |
         |     V             .--------.      | right_1
         |     <-----------+ AES-ECB |<-----+
         |     |             '________'      |
         |     | left_1  .--------.          V
         |     +-------->| AES-ECB +-------->
         |     |         '________'          |
         |     V             .--------.      |
         |     <-----------+ AES-ECB |<-----+
         |     |             '________'      |
         |     |                             |
         |     V                             V
         |  +-----------------+-----------------+
         |  |      left_2     |     right_2     |
         |  +-------+---------+--------+--------+
         |          |                  |
         V          V                  V
      +-----+-----------------------------------+
      | FO  |            Ciphertext             |
      +-----+-----------------------------------+
```

4.3.2.2. Useful functions

Two functions are useful to define:

The expand(length, pass, input_bytes) function concatenates three arguments and outputs 16 zero-padded octets.

The output of expand is as follows:

```
ExpandResult {
     input_bytes(...),
     ZeroPad(...),
     length(8),
     pass(8)
}
```

in which:

* 'input_bytes' is drawn from one half of the plaintext. It forms the N most significant octets of the output, where N is half the 'length' argument, rounded up, and thus a number between 3 and 10, inclusive.

* 'Zeropad' is a set of 14-N octets set to zero.

* 'length' is an 8-bit integer that reports the sum of the configured nonce length and server id length in octets, and forms the fifteenth octet of the output. The 'length' argument MUST NOT exceed 19 and MUST NOT be less than 5.

* 'pass' is an 8-bit integer that reports the 'pass' argument of the algorithm, and forms the sixteenth (least significant) octet of the output. It guarantees that the cryptographic input of every pass of the algorithm is unique.

For example,

```
expand(0x06, 0x02, 0xaaba3c) = 0xaaba3c00000000000000000000000602
```

Similarly, truncate(input, n) returns the first n octets of 'input'.

```
truncate(0x2094842ca49256198c2deaa0ba53caa0, 4) = 0x2094842c
```

Let 'half_len' be equal to 'plaintext_len' / 2, rounded up.

4.3.2.3. Algorithm Description

The example at the end of this section helps to clarify the steps described below.

1. The server concatenates the server ID and nonce to create plaintext_CID. The length of the result in octets is plaintext_len.

2. The server splits plaintext_CID into components left_0 and right_0 of equal length half_len. If plaintext_len is odd, right_0 clears its first four bits, and left_0 clears its last four bits. For example, 0x7040b81b55ccf3 would split into a left_0 of 0x7040b810 and right_0 of 0x0b55ccf3.

3. Encrypt the result of expand(plaintext_len, 1, left_0) using an AES-ECB-128 cipher to obtain a ciphertext.

4. XOR the first half_len octets of the ciphertext with right_0 to form right_1. Steps 3 and 4 can be summarized as

```
result = AES_ECB(key, expand(plaintext_len, 1, left_0))
right_1 = XOR(right_0, truncate(result, half_len))
```

5. If the plaintext_len is odd, clear the first four bits of right_1.

6. Repeat steps 3 and 4, but use them to compute left_1 by expanding and encrypting right_1 with pass = 2, and XOR the results with left_0.

```
result = AES_ECB(key, expand(plaintext_len, 2, right_1))
left_1 = XOR(left_0, truncate(result, half_len))
```

7. If the plaintext_len is odd, clear the last four bits of left_1.

8. Repeat steps 3 and 4, but use them to compute right_2 by expanding and encrypting left_1 with pass = 3, and XOR the results with right_1.

```
result = AES_ECB(key, expand(plaintext_len, 3, left_1))
right_2 = XOR(right_1, truncate(result, half_len))
```

9. If the plaintext_len is odd, clear the first four bits of right_2.

10. Repeat steps 3 and 4, but use them to compute left_2 by expanding and encrypting right_2 with pass = 4, and XOR the results with left_1.

```
result = AES_ECB(key, expand(plaintext_len, 4, right_2))
left_2 = XOR(left_1, truncate(result, half_len))
```

11. If the plaintext_len is odd, clear the last four bits of left_2.

12. The server concatenates left_2 with right_2 to form the ciphertext CID, which it appends to the first octet. If plaintext_len is odd, the four least significant bits of left_2 and four most significant bits of right_2, which are all zero, are stripped off before concatenation to make the resulting ciphertext the same length as the original plaintext.

4.3.2.4. Encryption Example

The following example executes the steps for the provided inputs. Note that the plaintext is of odd octet length, so the middle octet will be split evenly left_0 and right_0.

```
server_id = 0x31441a
nonce = 0x9c69c275
key = 0xfdf726a9893ec05c0632d3956680baf0

// step 1
plaintext_CID = 0x31441a9c69c275
plaintext_len = 7

// step 2
hash_len = 4
left_0 = 0x31441a90
right_0 = 0x0c69c275

// step 3
aes_input = 0x31441a90000000000000000000000701
aes_output = 0xa255dd8cdacf01948d3a848c3c7fee23

// step 4
right_1 = 0x0c69c275 ^ 0xa255dd8c = 0xae3c1ff9

// step 5 (clear bits)
right_1 = 0x0e8c1ff9

// step 6
aes_input = 0x0e8c1ff9000000000000000000000702
aes_output = 0xe5e452cb9e1bedb0b2bf830506bf4c4e
left_1 = 0x31441a90 ^ 0xe5e452cb = 0xd4a0485b

// step 7 (clear bits)
left_1 = 0xd4a04850

// step 8
aes_input = 0xd4a04850000000000000000000000703
aes_output = 0xb7821ab3024fed0913b6a04d18e3216f
right_2 = 0x0e8c1ff9 ^ 0xb7821ab3 = 0xb9be054a

// step 9 (clear bits)
right_2 = 0x09be054a

// step 10
aes_input = 0x09be054a000000000000000000000704
aes_output = 0xb334357cfdf81e3fafe180154eaf7378
```

```
left_2 = 0xd4a04850 ^ 0xb3e4357c = 0x67947d2c

// step 11 (clear bits)
left_2 = 0x67947d20

// step 12
cid = first_octet || left_2 || right_2 = 0x0767947d29be054a
```

4.4. Load Balancer Actions

On each incoming packet, the load balancer extracts consecutive octets, beginning with the second octet. If there is no key, the first octets correspond to the server ID.

If there is a key, the load balancer takes one of two actions:

4.4.1. Special Case: Single Pass Encryption

If server ID length and nonce length sum to exactly 16 octets, they form a ciphertext block. The load balancer decrypts the block using the AES-ECB key and extracts the server ID from the most significant bytes of the resulting plaintext.

4.4.2. General Case: Four-Pass Encryption

First, split the ciphertext CID (excluding the first octet) into its equal- length components left_2 and right_2. Then follow the process below:

```
result = AES_ECB(key, expand(plaintext_len, 4, right_2))
left_1 = XOR(left_2, truncate(result, half_len))
if (plaintext_len_is_odd()) clear_last_bits(left_1, 4)

result = AES_ECB(key, expand(plaintext_len, 3, left_1))
right_1 = XOR(right_2, truncate(result, half_len))
if (plaintext_len_is_odd()) clear_first_bits(left_1, 4)

result = AES_ECB(key, expand(plaintext_len, 2, right_1))
left_0 = XOR(left_1, truncate(result, half_len))
if (plaintext_len_is_odd()) clear_last_bits(left_0, 4)
```

As the load balancer has no need for the nonce, it can conclude after 3 passes as long as the server ID is entirely contained in left_0 (i.e., the nonce is at least as large as the server ID). If the server ID is longer, a fourth pass is necessary:

```
result = AES_ECB(key, expand(plaintext_len, 1, left_0))
right_0 = XOR(right_1, truncate(result, half_len))
if (plaintext_len_is_odd()) clear_first_bits(right_0, 4)
```

and the load balancer has to concatenate left_0 and right_0 to obtain the complete server ID.

5. Per-connection state

The CID allocation methods QUIC-LB defines require no per-connection state at the load balancer. The load balancer can extract the server ID from the connection ID of each incoming packet and route that packet accordingly.

However, once a routing decision has been made, the load balancer MAY associate the 4-tuple or connection ID with the decision. This has two advantages:

* The load balancer only extracts the server ID once until the 4-tuple or connection ID changes. When the CID is encrypted, this might reduce computational load.

* Incoming Stateless Reset packets and ICMP messages are easily routed to the correct origin server.

In addition to the increased state requirements, however, load balancers cannot detect the CONNECTION_CLOSE frame to indicate the end of the connection, so they rely on a timeout to delete connection state. There are numerous considerations around setting such a timeout.

In the event a connection ends, freeing an IP and port, and a different connection migrates to that IP and port before the timeout, the load balancer will misroute the different connection's packets to the original server. A short timeout limits the likelihood of such a misrouting.

Furthermore, if a short timeout causes premature deletion of state, the routing is easily recoverable by decoding an incoming Connection ID. However, a short timeout also reduces the chance that an incoming Stateless Reset is correctly routed.

Servers MAY implement the technique described in Section 14.4.1 of [RFC9000] in case the load balancer is stateless, to increase the likelihood a Source Connection ID is included in ICMP responses to Path Maximum Transmission Unit (PMTU) probes. Load balancers MAY parse the echoed packet to extract the Source Connection ID, if it contains a QUIC long header, and extract the Server ID as if it were in a Destination CID.

6. Additional Use Cases

This section discusses considerations for some deployment scenarios not implied by the specification above.

6.1. Load balancer chains

Some network architectures may have multiple tiers of low-state load balancers, where a first tier of devices makes a routing decision to the next tier, and so on, until packets reach the server. Although QUIC-LB is not explicitly designed for this use case, it is possible to support it.

If each load balancer is assigned a range of server IDs that is a subset of the range of IDs assigned to devices that are closer to the client, then the first devices to process an incoming packet can extract the server ID and then map it to the correct forwarding address. Note that this solution is extensible to arbitrarily large numbers of load-balancing tiers, as the maximum server ID space is quite large.

If the number of necessary server IDs per next hop is uniform, a simple implementation would use successively longer server IDs at each tier of load balancing, and the server configuration would match the last tier. Load balancers closer to the client can then treat any parts of the server ID they did not use as part of the nonce.

6.2. Server Process Demultiplexing

QUIC servers might have QUIC running on multiple processes listening on the same address, and have a need to demultiplex between them. In principle, this demultiplexer is a Layer 4 load balancer, and the guidance in Section 6.1 applies. However, in many deployments the demultiplexer lacks the capability to perform decryption operations. Internal server coordination is out of scope of this specification, but this non-normative section proposes some approaches that could work given certain server capabilities:

* Some bytes of the server ID are reserved to encode the process ID. The demultiplexer might operate based on the 4-tuple or other legacy indicator, but the receiving server process extracts the server ID, and if it does not match the one for that process, the process could "toss" the packet to the correct destination process.

* Each process could register the connection IDs it generates with the demultiplexer, which routes those connection IDs accordingly.

* In a combination of the two approaches above, the demultiplexer generally routes by 4-tuple. After a migration, the process tosses the first flight of packets and registers the new connection ID with the demultiplexer. This alternative limits the bandwidth consumption of tossing and the memory footprint of a full connection ID table.

* When generating a connection ID, the server writes the process ID to the random field of the first octet, or if this is being used for length encoding, in an octet it appends after the ciphertext. It then applies a keyed hash (with a key locally generated for the sole use of that server). The hash result is used as a bitmask to XOR with the bits encoding the process ID. On packet receipt, the demultiplexer applies the same keyed hash to generate the same mask and recoversthe process ID. (Note that this approach is conceptually similar to QUIC header protection).

6.3. Moving connections between servers

Some deployments may transparently move a connection from one server to another. The means of transferring connection state between servers is out of scope of this document.

To support a handover, a server involved in the transition could issue CIDs that map to the new server via a NEW_CONNECTION_ID frame, and retire CIDs associated with the old server using the "Retire Prior To" field in that frame.

7. Version Invariance of QUIC-LB

The server ID encodings, and requirements for their handling, are designed to be QUIC version independent (see [RFC8999]). A QUIC-LB load balancer will generally not require changes as servers deploy new versions of QUIC. However, there are several unlikely future design decisions that could impact the operation of QUIC-LB.

A QUIC version might define limits on connection ID length that make some or all of the mechanisms in this document unusable. For example, a maximum connection ID length could be below the minimum necessary to use all or part of this specification; or, the minimum connection ID length could be larger than the largest value in this specification.

Section 3.1 provides guidance about how load balancers should handle unroutable DCIDs. This guidance, and the implementation of an algorithm to handle these DCIDs, rests on some assumptions:

* Incoming short headers do not contain DCIDs that are client-generated.

* The use of client-generated incoming DCIDs does not persist beyond a few round trips in the connection.

* While the client is using DCIDs it generated, some exposed fields (IP address, UDP port, client-generated destination Connection ID) remain constant for all packets sent on the same connection.

While this document does not update the commitments in [RFC8999], the additional assumptions are minimal and narrowly scoped, and provide a likely set of constants that load balancers can use with minimal risk of version- dependence.

If these assumptions are not valid, this specification is likely to lead to loss of packets that contain unroutable DCIDs, and in extreme cases connection failure. A QUIC version that violates the assumptions in this section therefore cannot be safely deployed with a load balancer that follows this specification. An updated or alternative version of this specification might address these shortcomings for such a QUIC version.

Some load balancers might inspect version-specific elements of packets to make a routing decision. This might include the Server Name Indication (SNI) extension in the TLS Client Hello. The format and cryptographic protection of this information may change in future versions or extensions of TLS or QUIC, and therefore this functionality is inherently not version-invariant. Such a load balancer, when it receives packets from an unknown QUIC version, might misdirect initial packets to the wrong tenant. While this can be inefficient, the design in this document preserves the ability for tenants to deploy new versions provided they have an out-of-band means of providing a connection ID for the client to use.

8. Security Considerations

QUIC-LB is intended to prevent linkability. Attacks would therefore attempt to subvert this purpose.

Note that without a key for the encoding, QUIC-LB makes no attempt to obscure the server mapping, and therefore does not address these concerns. Without a key, QUIC-LB merely allows consistent CID encoding for compatibility across a network infrastructure, which makes QUIC robust to NAT rebinding. Servers that are encoding their server ID without a key algorithm SHOULD only use it to generate new CIDs for the Server Initial Packet and SHOULD NOT send CIDs in QUIC NEW_CONNECTION_ID frames, except that it sends one new Connection ID in the event of config rotation Section 2.1. Doing so might falsely suggest to the client that said CIDs were generated in a secure fashion.

A linkability attack would find some means of determining that two connection IDs route to the same server. Due to the limitations of measures at QUIC layer, there is no scheme that strictly prevents linkability for all traffic patterns.

To see why, consider two limits. At one extreme, one client is connected to the server pool and migrates its address. An observer can easily link the two addresses, and there is no remedy at the QUIC layer.

At the other extreme, a very large number of clients are connected to each server, and they all migrate address constantly. At this limit, even an unencrypted server ID encoding is unlikely to definitively link two addresses.

Therefore, efforts to frustrate any analysis of server ID encoding have diminishing returns. Nevertheless, this specification seeks to minimize the probability two addresses can be linked.

8.1. Attackers not between the load balancer and server

Any attacker might open a connection to the server infrastructure and aggressively simulate migration to obtain a large sample of IDs that map to the same server. It could then apply analytical techniques to try to obtain the server encoding.

An encrypted encoding provides robust protection against this. An unencrypted one provides none.

Were this analysis to obtain the server encoding, then on-path observers might apply this analysis to correlating different client IP addresses.

8.2. Attackers between the load balancer and server

Attackers in this privileged position are intrinsically able to map two connection IDs to the same server. These algorithms ensure that two connection IDs for the same connection cannot be identified as such as long as the server chooses the first octet and any plaintext nonce correctly.

8.3. Multiple Configuration IDs

During the period in which there are multiple deployed configuration IDs (see Section 2.1), there is a slight increase in linkability. The server space is effectively divided into segments with CIDs that have different config rotation bits. Entities that manage servers SHOULD strive to minimize these periods by quickly deploying new configurations across the server pool.

8.4. Limited configuration scope

A simple deployment of QUIC-LB in a cloud provider might use the same global QUIC-LB configuration across all its load balancers that route to customer servers. An attacker could then simply become a customer, obtain the configuration, and then extract server IDs of other customers' connections at will.

To avoid this, the configuration agent SHOULD issue QUIC-LB configurations to mutually distrustful servers that have different keys for encryption algorithms. In many cases, the load balancers can distinguish these configurations by external IP address.

However, assigning multiple entities to an IP address is complimentary with concealing DNS requests (e.g., DoH [RFC8484]) and the TLS Server Name Indicator (SNI) ([I-D.ietf-tls-esni]) to obscure the ultimate destination of traffic. While the load balancer's fallback algorithm (Section 3.2) can use the SNI to make a routing decision on the first packet, there are three ways to route subsequent packets:

* all co-tenants can use the same QUIC-LB configuration, leaking the server mapping to each other as described above;

* co-tenants can be issued one of up to seven configurations distinguished by the config rotation bits (Section 2.1), exposing information about the target domain to the entire network; or

* tenants can use the 0b111 codepoint in their CIDs (in which case they SHOULD disable migration in their connections), which neutralizes the value of QUIC-LB but preserves privacy.

When configuring QUIC-LB, administrators evaluate the privacy tradeoff by considering the relative value of each of these properties, given the trust model between tenants, the presence of methods to obscure the domain name, and value of address migration in the tenant use cases.

As the plaintext algorithm makes no attempt to conceal the server mapping, these deployments MAY simply use a common configuration.

8.5. Stateless Reset Oracle

Section 21.9 of [RFC9000] discusses the Stateless Reset Oracle attack. For a server deployment to be vulnerable, an attacking client must be able to cause two packets with the same Destination CID to arrive at two different servers that share the same cryptographic context for Stateless Reset tokens. As QUIC-LB requires deterministic routing of DCIDs over the life of a connection, it is a sufficient means of avoiding an Oracle without additional measures.

Note also that when a server starts using a new QUIC-LB config rotation codepoint, new CIDs might not be unique with respect to previous configurations that occupied that codepoint, and therefore different clients may have observed the same CID and stateless reset token. A straightforward method of managing stateless reset keys is to maintain a separate key for each config rotation codepoint, and replace each key when the configuration for that codepoint changes. Thus, a server transitions from one config to another, it will be able to generate correct tokens for connections using either type of CID.

8.6. Connection ID Entropy

If a server ever reuses a nonce in generating a CID for a given configuration, it risks exposing sensitive information. Given the same server ID, the CID will be identical (aside from a possible difference in the first octet). This can risk exposure of the QUIC-LB key. If two clients receive the same connection ID, they also have each other’s stateless reset token unless that key has changed in the interim.

The encrypted mode needs to generate different cipher text for each generated Connection ID instance to protect the Server ID. To do so, at least four octets of the CID are reserved for a nonce that, if used only once, will result in unique cipher text for each Connection ID.

If servers simply increment the nonce by one with each generated connection ID, then it is safe to use the existing keys until any server's nonce counter exhausts the allocated space and rolls over. To maximize entropy, servers SHOULD start with a random nonce value, in which case the configuration is usable until the nonce value wraps around to zero and then reaches the initial value again.

Whether or not it implements the counter method, the server MUST NOT reuse a nonce until it switches to a configuration with new keys.

Servers are forbidden from generating linkable plaintext nonces, because observable correlations between plaintext nonces would provide trivial linkability between individual connections, rather than just to a common server.

For any algorithm, configuration agents SHOULD implement an out-of-band method to discover when servers are in danger of exhausting their nonce space, and SHOULD respond by issuing a new configuration. A server that has exhausted its nonces MUST either switch to a different configuration, or if none exists, use the 4-tuple routing config rotation codepoint.

When sizing a nonce that is to be randomly generated, the configuration agent SHOULD consider that a server generating a N-bit nonce will create a duplicate about every 2^(N/2) attempts, and therefore compare the expected rate at which servers will generate CIDs with the lifetime of a configuration.

8.7. Distinguishing Attacks

The Four Pass Encryption algorithm is structured as a 4-round Feistel network with non-bijective round function. As such, it does not offer a very high security level against distinguishing attacks, as explained in [Patarin2008]. Attackers can mount these attacks if they are in possession of O(SQRT(len/2)) pairs of ciphertext and known corresponding plain text, where "len" is the sum of the lengths of the Server ID and the Nonce.

The authors considered increasing the number of passes from 4 to 12, which would definitely block these attacks. However, this would require 12 round of AES decryption by load balancers accessing the CID, a cost deemed prohibitive in the planned deployments.

The attacks described in [Patarin2008] rely on known plain text. In a normal deployment, the plain text is only known by the server that generates the ID and by the load balancer that decrypts the content of the CID. Attackers would have to compensate by guesses about the allocation of server identifiers or the generation of nonces. These attacks are thus mitigated by making nonces hard to guess, as specified in Section 8.6, and by rules related to mixed deployments that use both clear text CID and encrypted CID, for example when transitioning from clear text to encryption. Such deployments MUST use different server ID allocations for the clear text and the encrypted versions.

These attacks cannot be mounted against the Single Pass Encryption algorithm.

9. IANA Considerations

There are no IANA requirements.

10. References

10.1. Normative References

[NIST-AES-ECB]
Dworkin, M., "Recommendation for Block Cipher Modes of Operation: Methods and Techniques", NIST Special Publication 800-38A, 2021, <https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-38a.pdf>.

[RFC8999] Thomson, M., "Version-Independent Properties of QUIC", RFC 8999, DOI 10.17487/RFC8999, May 2021, <https://www.rfc-editor.org/rfc/rfc8999>.

[RFC9000] Iyengar, J., Ed. and M. Thomson, Ed., "QUIC: A UDP-Based Multiplexed and Secure Transport", RFC 9000, DOI 10.17487/RFC9000, May 2021, <https://www.rfc-editor.org/rfc/rfc9000>.

10.2. Informative References

[I-D.ietf-tls-esni]
Rescorla, E., Oku, K., Sullivan, N., and C. A. Wood, "TLS Encrypted Client Hello", Work in Progress, Internet-Draft, draft-ietf-tls-esni-17, 9 October 2023, <https://datatracker.ietf.org/doc/html/draft-ietf-tls-esni-17>.

[Patarin2008]
Patarin, J., "Generic Attacks on Feistel Schemes - Extended Version", 2008, <https://eprint.iacr.org/2008/036.pdf>.

[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/rfc/rfc2119>.

[RFC4347] Rescorla, E. and N. Modadugu, "Datagram Transport Layer Security", RFC 4347, DOI 10.17487/RFC4347, April 2006, <https://www.rfc-editor.org/rfc/rfc4347>.

[RFC6020] Bjorklund, M., Ed., "YANG - A Data Modeling Language for the Network Configuration Protocol (NETCONF)", RFC 6020, DOI 10.17487/RFC6020, October 2010, <https://www.rfc-editor.org/rfc/rfc6020>.

[RFC6347] Rescorla, E. and N. Modadugu, "Datagram Transport Layer Security Version 1.2", RFC 6347, DOI 10.17487/RFC6347, January 2012, <https://www.rfc-editor.org/rfc/rfc6347>.

[RFC7696] Housley, R., "Guidelines for Cryptographic Algorithm Agility and Selecting Mandatory-to-Implement Algorithms", BCP 201, RFC 7696, DOI 10.17487/RFC7696, November 2015, <https://www.rfc-editor.org/rfc/rfc7696>.

[RFC7983] Petit-Huguenin, M. and G. Salgueiro, "Multiplexing Scheme Updates for Secure Real-time Transport Protocol (SRTP) Extension for Datagram Transport Layer Security (DTLS)", RFC 7983, DOI 10.17487/RFC7983, September 2016, <https://www.rfc-editor.org/rfc/rfc7983>.

[RFC8340] Bjorklund, M. and L. Berger, Ed., "YANG Tree Diagrams", BCP 215, RFC 8340, DOI 10.17487/RFC8340, March 2018, <https://www.rfc-editor.org/rfc/rfc8340>.

[RFC8484] Hoffman, P. and P. McManus, "DNS Queries over HTTPS (DoH)", RFC 8484, DOI 10.17487/RFC8484, October 2018, <https://www.rfc-editor.org/rfc/rfc8484>.

[RFC9146] Rescorla, E., Ed., Tschofenig, H., Ed., Fossati, T., and A. Kraus, "Connection Identifier for DTLS 1.2", RFC 9146, DOI 10.17487/RFC9146, March 2022, <https://www.rfc-editor.org/rfc/rfc9146>.

[RFC9147] Rescorla, E., Tschofenig, H., and N. Modadugu, "The Datagram Transport Layer Security (DTLS) Protocol Version 1.3", RFC 9147, DOI 10.17487/RFC9147, April 2022, <https://www.rfc-editor.org/rfc/rfc9147>.

Appendix A. QUIC-LB YANG Model

These YANG models conform to [RFC6020] and express a complete QUIC-LB configuration. There is one model for the server and one for the middlebox (i.e the load balancer and/or Retry Service).

```
module ietf-quic-lb-server {
  yang-version "1.1";
  namespace "urn:ietf:params:xml:ns:yang:ietf-quic-lb";
  prefix "quic-lb";

  import ietf-yang-types {
    prefix yang;
    reference
      "RFC 6991: Common YANG Data Types.";
  }

  import ietf-inet-types {
    prefix inet;
    reference
      "RFC 6991: Common YANG Data Types.";
  }

  organization
    "IETF QUIC Working Group";

  contact
    "WG Web:   <http://datatracker.ietf.org/wg/quic>
     WG List:  <quic@ietf.org>

     Authors: Martin Duke (martin.h.duke at gmail dot com)
              Nick Banks (nibanks at microsoft dot com)
              Christian Huitema (huitema at huitema.net)";

  description
    "This module enables the explicit cooperation of QUIC servers
     with trusted intermediaries without breaking important
     protocol features.

     Copyright (c) 2022 IETF Trust and the persons identified as
     authors of the code.  All rights reserved.

     Redistribution and use in source and binary forms, with or
```

```
         without modification, is permitted pursuant to, and subject to
         the license terms contained in, the Simplified BSD License set
         forth in Section 4.c of the IETF Trust's Legal Provisions
         Relating to IETF Documents
         (https://trustee.ietf.org/license-info).

         This version of this YANG module is part of RFC XXXX
         (https://www.rfc-editor.org/info/rfcXXXX); see the RFC itself
         for full legal notices.

         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 (RFC 2119) (RFC 8174) when, and only when,
         they appear in all capitals, as shown here.";

      revision "2023-07-14" {
        description
          "Updated to design in version 17 of the draft";
        reference
          "RFC XXXX, QUIC-LB: Generating Routable QUIC Connection IDs";
      }

      container quic-lb {
        presence "The container for QUIC-LB configuration.";

        description
          "QUIC-LB container.";

        typedef quic-lb-key {
          type yang:hex-string {
            length 47;
          }
          description
            "This is a 16-byte key, represented with 47 bytes";
        }

        leaf config-id {
          type uint8 {
            range "0..6";
          }
          mandatory true;
          description
            "Identifier for this CID configuration.";
        }

        leaf first-octet-encodes-cid-length {
          type boolean;
```

```
          default false;
          description
            "If true, the six least significant bits of the first
             CID octet encode the CID length minus one.";
        }

        leaf server-id-length {
          type uint8 {
            range "1..15";
          }
          must '. <= (19 - ../nonce-length)' {
            error-message
              "Server ID and nonce lengths must sum
               to no more than 19.";
          }
          mandatory true;
          description
            "Length (in octets) of a server ID. Further range-limited
             by nonce-length.";
        }

        leaf nonce-length {
          type uint8 {
            range "4..18";
          }
          mandatory true;
          description
            "Length, in octets, of the nonce. Short nonces mean there
             will be frequent configuration updates.";
        }

        leaf cid-key {
          type quic-lb-key;
          description
            "Key for encrypting the connection ID.";
        }

        leaf server-id {
          type yang:hex-string;
          must "string-length(.) = 3 * ../../server-id-length - 1";
          mandatory true;
          description
            "An allocated server ID";
        }
      }
    }
```

```
module ietf-quic-lb-middlebox {
  yang-version "1.1";
  namespace "urn:ietf:params:xml:ns:yang:ietf-quic-lb";
  prefix "quic-lb";

  import ietf-yang-types {
    prefix yang;
    reference
      "RFC 6991: Common YANG Data Types.";
  }

  import ietf-inet-types {
    prefix inet;
    reference
      "RFC 6991: Common YANG Data Types.";
  }

  organization
    "IETF QUIC Working Group";

  contact
    "WG Web:   <http://datatracker.ietf.org/wg/quic>
     WG List:  <quic@ietf.org>

     Authors: Martin Duke (martin.h.duke at gmail dot com)
              Nick Banks (nibanks at microsoft dot com)
              Christian Huitema (huitema at huitema.net)";

  description
    "This module enables the explicit cooperation of QUIC servers
     with trusted intermediaries without breaking important
     protocol features.

     Copyright (c) 2021 IETF Trust and the persons identified as
     authors of the code.  All rights reserved.

     Redistribution and use in source and binary forms, with or
     without modification, is permitted pursuant to, and subject to
     the license terms contained in, the Simplified BSD License set
     forth in Section 4.c of the IETF Trust's Legal Provisions
     Relating to IETF Documents
     (https://trustee.ietf.org/license-info).

     This version of this YANG module is part of RFC XXXX
     (https://www.rfc-editor.org/info/rfcXXXX); see the RFC itself
     for full legal notices.

     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 (RFC 2119) (RFC 8174) when, and only when,
         they appear in all capitals, as shown here.";

      revision "2021-02-11" {
        description
          "Updated to design in version 13 of the draft";
        reference
          "RFC XXXX, QUIC-LB: Generating Routable QUIC Connection IDs";
      }

      container quic-lb {
        presence "The container for QUIC-LB configuration.";

        description
          "QUIC-LB container.";

        typedef quic-lb-key {
          type yang:hex-string {
            length 47;
          }
          description
            "This is a 16-byte key, represented with 47 bytes";
        }

        list cid-configs {
          key "config-rotation-bits";
          description
            "List up to three load balancer configurations";

          leaf config-rotation-bits {
            type uint8 {
              range "0..2";
            }
            mandatory true;
            description
              "Identifier for this CID configuration.";
          }

          leaf server-id-length {
            type uint8 {
              range "1..15";
            }
            must '. <= (19 - ../nonce-length)' {
              error-message
                "Server ID and nonce lengths must sum to
                 no more than 19.";
```

```
          }
          mandatory true;
          description
            "Length (in octets) of a server ID. Further range-limited
             by nonce-length.";
        }

        leaf cid-key {
          type quic-lb-key;
          description
            "Key for encrypting the connection ID.";
        }

        leaf nonce-length {
          type uint8 {
            range "4..18";
          }
          mandatory true;
          description
            "Length, in octets, of the nonce. Short nonces mean there
             will be frequent configuration updates.";
        }

        list server-id-mappings {
          key "server-id";
          description "Statically allocated Server IDs";

          leaf server-id {
            type yang:hex-string;
            must "string-length(.) = 3 * ../../server-id-length - 1";
            mandatory true;
            description
              "An allocated server ID";

          }

          leaf server-address {
            type inet:ip-address;
            mandatory true;
            description
              "Destination address corresponding to the server ID";
          }
        }
      }
    }
  }
```

A.1. Tree Diagram

This summary of the YANG models uses the notation in [RFC8340].

```
module: ietf-quic-lb-server
  +--rw quic-lb!
     +--rw config-id                        uint8
     +--rw first-octet-encodes-cid-length?   boolean
     +--rw server-id-length                 uint8
     +--rw nonce-length                     uint8
     +--rw cid-key?                         quic-lb-key
     +--rw server-id                        yang:hex-string

module: ietf-quic-lb-middlebox
  +--rw quic-lb!
     +--rw cid-configs* [config-rotation-bits]
     |  +--rw config-rotation-bits    uint8
     |  +--rw server-id-length        uint8
     |  +--rw cid-key?                quic-lb-key
     |  +--rw nonce-length            uint8
     |  +--rw server-id-mappings* [server-id]
     |     +--rw server-id         yang:hex-string
     |     +--rw server-address    inet:ip-address
```

Appendix B. Load Balancer Test Vectors

This section uses the following abbreviations:

```
cid      Connection ID
cr_bits  Config Rotation Bits
LB       Load Balancer
sid      Server ID
```

In all cases, the server is configured to encode the CID length.

B.1. Unencrypted CIDs

```
cr_bits sid nonce cid
0 c4605e 4504cc4f 07c4605e4504cc4f
1 350d28b420 3487d970b 20a350d28b4203487d970b
```

B.2. Encrypted CIDs

The key for all of these examples is 8f95f09245765f80256934e50c66207f. The test vectors include an example that uses the 16-octet single-pass special case, as well as an instance where the server ID length exceeds the nonce length, requiring a fourth decryption pass.

```
cr_bits sid nonce cid
0 ed793a ee080dbf 0720b1d07b359d3c
1 ed793a51d49b8f5fab65 ee080dbf48
                         2fcc381bc74cb4fbad2823a3d1f8fed2
2 ed793a51d49b8f5f ee080dbf48c0d1e5
                         504dd2d05a7b0de9b2b9907afb5ecf8cc3
3 ed793a51d49b8f5fab ee080dbf48c0d1e55d
                         125779c9cc86beb3a3a4a3ca96fce4bfe0cdbc
```

Appendix C. Interoperability with DTLS over UDP

Some environments may contain DTLS traffic as well as QUIC operating over UDP, which may be hard to distinguish.

In most cases, the packet parsing rules above will cause a QUIC-LB load balancer to route DTLS traffic in an appropriate way. DTLS 1.3 implementations that use the connection_id extension [RFC9146] might use the techniques in this document to generate connection IDs and achieve robust routability for DTLS associations if they meet a few additional requirements. This non-normative appendix describes this interaction.

C.1. DTLS 1.0 and 1.2

DTLS 1.0 [RFC4347] and 1.2 [RFC6347] use packet formats that a QUIC-LB router will interpret as short header packets with CIDs that request 4-tuple routing. As such, they will route such packets consistently as long as the 4-tuple does not change. Note that DTLS 1.0 has been deprecated by the IETF.

The first octet of every DTLS 1.0 or 1.2 datagram contains the content type. A QUIC-LB load balancer will interpret any content type less than 128 as a short header packet, meaning that the subsequent octets should contain a connection ID.

Existing TLS content types comfortably fit in the range below 128. Assignment of codepoints greater than 64 would require coordination in accordance with [RFC7983], and anyway would likely create problems demultiplexing DTLS and version 1 of QUIC. Therefore, this document believes it is extremely unlikely that TLS content types of 128 or greater will be assigned. Nevertheless, such an assignment would cause a QUIC-LB load balancer to interpret the packet as a QUIC long header with an essentially random connection ID, which is likely to be routed irregularly.

The second octet of every DTLS 1.0 or 1.2 datagram is the bitwise complement of the DTLS Major version (i.e. version 1.x = 0xfe). A QUIC-LB load balancer will interpret this as a connection ID that requires 4-tuple based load balancing, meaning that the routing will be consistent as long as the 4-tuple remains the same.

[RFC9146] defines an extension to add connection IDs to DTLS 1.2. Unfortunately, a QUIC-LB load balancer will not correctly parse the connection ID and will continue 4-tuple routing. An modified QUIC-LB load balancer that correctly identifies DTLS and parses a DTLS 1.2 datagram for the connection ID is outside the scope of this document.

C.2. DTLS 1.3

DTLS 1.3 [RFC9147] changes the structure of datagram headers in relevant ways.

Handshake packets continue to have a TLS content type in the first octet and 0xfe in the second octet, so they will be 4-tuple routed, which should not present problems for likely NAT rebinding or address change events.

Non-handshake packets always have zero in their most significant bit and will therefore always be treated as QUIC short headers. If the connection ID is present, it follows in the succeeding octets. Therefore, a DTLS 1.3 association where the server utilizes Connection IDs and the encodings in this document will be routed correctly in the presence of client address and port changes.

However, if the client does not include the connection_id extension in its ClientHello, the server is unable to use connection IDs. In this case, non- handshake packets will appear to contain random connection IDs and be routed randomly. Thus, unmodified QUIC-LB load balancers will not work with DTLS 1.3 if the client does not advertise support for connection IDs, or the server does not request the use of a compliant connection ID.

A QUIC-LB load balancer might be modified to identify DTLS 1.3 packets and correctly parse the fields to identify when there is no connection ID and revert to 4-tuple routing, removing the server requirement above. However, such a modification is outside the scope of this document, and classifying some packets as DTLS might be incompatible with future versions of QUIC.

C.3. Future Versions of DTLS

As DTLS does not have an IETF consensus document that defines what parts of DTLS will be invariant in future versions, it is difficult to speculate about the applicability of this section to future versions of DTLS.

Appendix D. Acknowledgments

Manasi Deval, Erik Fuller, Toma Gavrichenkov, Jana Iyengar, Subodh Iyengar, Stefan Kolbl, Ladislav Lhotka, Jan Lindblad, Ling Tao Nju, Ilari Liusvaara, Kazuho Oku, Udip Pant, Ian Swett, Andy Sykes, Martin Thomson, Dmitri Tikhonov, Victor Vasiliev, and William Zeng Ke all provided useful input to this document.

Appendix E. Change Log

> *RFC Editor's Note:* Please remove this section prior to publication of a final version of this document.

E.1. since draft-ietf-quic-load-balancers-18

* Rearranged the output of the expand function to reduce CPU load of decrypt

E.2. since draft-ietf-quic-load-balancers-17

* fixed regressions in draft-17 publication

E.3. since draft-ietf-quic-load-balancers-16

* added a config ID bit (now there are 3).

E.4. since draft-ietf-quic-load-balancers-15

* aasvg fixes.

E.5. since draft-ietf-quic-load-balancers-14

* Revised process demultiplexing text

* Restored lost text in Security Considerations

* Editorial comments from Martin Thomson.

* Tweaked 4-pass algorithm to avoid accidental plaintext similarities

E.6. since draft-ietf-quic-load-balancers-13

* Incorporated Connection ID length in argument of truncate function

* Added requirements for codepoint 0b11.

* Describe Distinguishing Attack in Security Considerations.

* Added non-normative language about server process demultiplexers

E.7. since draft-ietf-quic-load-balancers-12

* Separated Retry Service design into a separate draft

E.8. since draft-ietf-quic-load-balancers-11

* Fixed mistakes in test vectors

E.9. since draft-ietf-quic-load-balancers-10

* Refactored algorithm descriptions; made the 4-pass algorithm easier to implement

* Revised test vectors

* Split YANG model into a server and middlebox version

E.10. since draft-ietf-quic-load-balancers-09

* Renamed "Stream Cipher" and "Block Cipher" to "Encrypted Short" and "Encrypted Long"

* Added section on per-connection state

* Changed "Encrypted Short" to a 4-pass algorithm.

* Recommended a random initial nonce when incrementing.

* Clarified what SNI LBs should do with unknown QUIC versions.

E.11. since draft-ietf-quic-load-balancers-08

* Eliminate Dynamic SID allocation

* Eliminated server use bytes

E.12. since draft-ietf-quic-load-balancers-07

* Shortened SSCID nonce minimum length to 4 bytes

* Removed RSCID from Retry token body

* Simplified CID formats

* Shrunk size of SID table

E.13. since draft-ietf-quic-load-balancers-06

* Added interoperability with DTLS

* Changed "non-compliant" to "unroutable"

* Changed "arbitrary" algorithm to "fallback"

* Revised security considerations for mistrustful tenants

* Added retry service considerations for non-Initial packets

E.14. since draft-ietf-quic-load-balancers-05

* Added low-config CID for further discussion

* Complete revision of shared-state Retry Token

* Added YANG model

* Updated configuration limits to ensure CID entropy

* Switched to notation from quic-transport

E.15. since draft-ietf-quic-load-balancers-04

* Rearranged the shared-state retry token to simplify token processing

* More compact timestamp in shared-state retry token

* Revised server requirements for shared-state retries

* Eliminated zero padding from the test vectors

* Added server use bytes to the test vectors

* Additional compliant DCID criteria

E.16. since-draft-ietf-quic-load-balancers-03

* Improved Config Rotation text

* Added stream cipher test vectors

* Deleted the Obfuscated CID algorithm

E.17. since-draft-ietf-quic-load-balancers-02

* Replaced stream cipher algorithm with three-pass version

* Updated Retry format to encode info for required TPs

* Added discussion of version invariance

* Cleaned up text about config rotation

* Added Reset Oracle and limited configuration considerations

* Allow dropped long-header packets for known QUIC versions

E.18. since-draft-ietf-quic-load-balancers-01

* Test vectors for load balancer decoding

* Deleted remnants of in-band protocol

* Light edit of Retry Services section

* Discussed load balancer chains

E.19. since-draft-ietf-quic-load-balancers-00

* Removed in-band protocol from the document

E.20. Since draft-duke-quic-load-balancers-06

* Switch to IETF WG draft.

E.21. Since draft-duke-quic-load-balancers-05

* Editorial changes

* Made load balancer behavior independent of QUIC version

* Got rid of token in stream cipher encoding, because server might not have it

* Defined "non-compliant DCID" and specified rules for handling them.

* Added psuedocode for config schema

E.22. Since draft-duke-quic-load-balancers-04

* Added standard for retry services

E.23. Since draft-duke-quic-load-balancers-03

* Renamed Plaintext CID algorithm as Obfuscated CID

* Added new Plaintext CID algorithm

* Updated to allow 20B CIDs

* Added self-encoding of CID length

E.24. Since draft-duke-quic-load-balancers-02

* Added Config Rotation

* Added failover mode

* Tweaks to existing CID algorithms

* Added Block Cipher CID algorithm

* Reformatted QUIC-LB packets

E.25. Since draft-duke-quic-load-balancers-01

* Complete rewrite

* Supports multiple security levels

* Lightweight messages

E.26. Since draft-duke-quic-load-balancers-00

* Converted to markdown

* Added variable length connection IDs

Authors' Addresses

Martin Duke
Google
Email: martin.h.duke@gmail.com

Nick Banks
Microsoft
Email: nibanks@microsoft.com

Christian Huitema
Private Octopus Inc.
Email: huitema@huitema.net