QUIC
Internet-Draft
Intended status: Standards Track
Expires: 26 November 2022

M. Duke
Google
N. Banks
Microsoft

25 May 2022

QUIC Retry Offload
draft-ietf-quic-retry-offload-00

Abstract

QUIC uses Retry packets to reduce load on stressed servers, by forcing the client to prove ownership of its address before the server commits state. QUIC also has an anti-tampering mechanism to prevent the unauthorized injection of Retry packets into a connection. However, a server operator may want to offload production of Retry packets to an anti-Denial-of-Service agent or hardware accelerator. "Retry Offload" is a mechanism for coordination between a server and an external generator of Retry packets that can succeed despite the anti-tampering mechanism.

Table of Contents

1. Introduction

QUIC [RFC9000] servers send Retry packets to avoid prematurely allocating resources when under stress, such as during a Denial of Service (DoS) attack. Because both Initial packets and Retry packets have weak authentication properties, the Retry packet contains an encrypted token that helps the client and server to validate, via transport parameters, that an attacker did not inject or modify a packet of either type for this connection attempt.

However, a server under stress is less inclined to process incoming Initial packets and compute the Retry token in the first place. An analogous mechanism for TCP is syncookies [RFC4987]. As TCP has weaker authentication properties to QUIC, syncookie generation can often be offloaded to a hardware device, or to a anti-Denial-of-Service provider that is topologically far from the protected server. As such an offload would behave exactly like an attacker, QUIC's authentication methods make such a capability impossible.

This document seeks to enable offloading of Retry generation to QUIC via explicit coordination between servers and the hardware or provider offload, which this document refers to as a "Retry Offload." It has two different modes, to conform to two different use cases.

The no-shared-state mode has minimal coordination and does not require key sharing. While operationally easier to configure and

manage, it places severe constraints on the operational profile of the offload. In particular, the offload must control all ingress to the server and fail closed.

The shared-state mode removes the operational constraints, but also requires more sophisticated key management.

Both modes specify a common format for encoding information in the Retry token, so that the server can correctly populate the relevant transport parameter fields.

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.

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

A "Retry Offload" is a hardware or software device that is conceptually separate from a QUIC server that terminates QUIC connections. This document assumes that the Retry Offload and the server have an administrative relationship that allows them to accept common configuation.

A "configuration agent" is some entity that determines the common configuration to be distributed to the servers and the Retry Offload.

This document uses "QUIC" to refer to the protocol in QUIC version 1 [RFC9000]. Retry offloads can be applied to other versions of QUIC that use Retry packets and have identical information requirements for Retry validation. However, note that source and destination connection IDs are the only relevant data fields that are invariant across QUIC versions [RFC8999].

1.2. Notation

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

The example below illustrates the basic framework:

```
Example Structure {
  One-bit Field (1),
  7-bit Field with Fixed Value (7) = 61,
  Field with Variable-Length Integer (i),
  Arbitrary-Length Field (..),
  Variable-Length Field (8..24),
  Field With Minimum Length (16..),
  Field With Maximum Length (..128),
  [Optional Field (64)],
  Repeated Field (8) ...,
}
```

Figure 1: Example Format

2. Common Requirements

Regardless of mechanism, a Retry Offload has an active mode, where it is generating Retry packets, and an inactive mode, where it is not, based on its assessment of server load and the likelihood an attack is underway. The choice of mode MAY be made on a per-packet or per-connection basis, through a stochastic process or based on client address.

A configuration agent MUST distribute a list of QUIC versions the Retry Offload supports. It MAY also distribute either an "Allow-List" or a "Deny-List" of other QUIC versions. It MUST NOT distribute both an Allow-List and a Deny-List.

The Allow-List or Deny-List MUST NOT include any versions included for Retry Offload support.

The Configuration Agent MUST provide a means for the entity that controls the Retry Offload to report its supported version(s) to the configuration Agent. If the entity has not reported this information, it MUST NOT activate the Retry Offload and the configuration agent MUST NOT distribute configuration that activates

it.

The configuration agent MAY delete versions from the final supported version list if policy does not require the Retry Offload to operate on those versions.

The configuration Agent MUST provide a means for the entities that control servers behind the Retry Offload to report either an Allow-List or a Deny-List.

If all entities supply Allow-Lists, the consolidated list MUST be the union of these sets. If all entities supply Deny-Lists, the consolidated list MUST be the intersection of these sets.

If entities provide a mixture of Allow-Lists and Deny-Lists, the consolidated list MUST be a Deny-List that is the intersection of all provided Deny-Lists and the inverses of all Allow-Lists.

If no entities that control servers have reported Allow-Lists or Deny-Lists, the default is a Deny-List with the null set (i.e., all unsupported versions will be admitted). This preserves the future extensibilty of QUIC.

A Retry Offload MUST forward all packets for a QUIC version it does not support that are not on a Deny-List or absent from an Allow-List. Note that if servers support versions the Retry Offload does not, this may increase load on the servers.

Note that future versions of QUIC might not have Retry packets, require different information in Retry, or use different packet type indicators.

2.1. Consistent Treatment of Initials

Retry Offloads SHOULD treat Initial packets from the same connection with a uniform policy. Initial packets of the first and second client flight can be difficult to distinguish without expensive decryption of the contents, which is unsuitable under the conditions

of a DDoS attack. If the first packet of a connection is admitted without Retry, but the second triggers a Retry, that Retry packet will be ignored and the loss of an Initial coalesced with other packets can impair performance. In some situations, the client does not yet have handshake keys, and dropping further client Initial packets creates a deadlock where the connection cannot progress.

The simplest means to ensure this is to require, when active, a Retry Token for all incoming Initial packets, and send a Retry packet otherwise. If the Retry Offload is to be more selective, one technique keeps state on which address/port 4-tuples have been admitted. Another would be to apply a secure hash to the source IP address, port, and connection ID to deterministically compute whether the Initial requires a Retry Token or not. These source values remain consistent over the handshake.

However, even with these techniques there is a potential problem when a Retry Offload switches from inactive to active mode. The Retry Offload could admit the first packet while in inactive mode, and then drop subsequent Initials in active mode.

If the Retry Offload is always on-path, it MAY keep state on incoming connections while in inactive mode to avoid this problem. If it cannot or will not keep such state, it SHOULD implement "transition mode" for an interval chosen to include the likely Initial packet exchange of most clients (200ms is a sensible default).

In transition mode, Retry Offloads process Initial packets with Retry tokens as in active mode. When the Retry Offload receives an Initial packet with no token, it issues a Retry AND forwards the packet to the server. If the client has already received a packet from the server, it will ignore the Retry and the connection will progress normally. If not, the client will reconnect based on the Retry, the

server's response to the first initial will be discarded, and the connection will progress normally based on the client's second Initial. Appendix B explores the various possible packet sequences in transition mode.

Note that transition mode provides no actual DDoS relief to the server, so its duration should be as short as possible. The Retry Offload can choose not to implement transition mode and cause some

client connections to fail.

Servers operating behind a Retry Offload SHOULD implement a mechanism that operates whenever a client Initial arrives with a valid Retry token. If there is another connection with identical client Connection ID, IP, and Port, but with an unvalidated address, that connection is immediately and silently terminated. This mechanism eliminates incorrect connection state that is an artifact of transition mode, as explained in Appendix B.

2.2. Considerations for Non-Initial Packets

Initial Packets are especially effective at consuming server resources because they cause the server to create connection state. Even when mitigating this load with Retry Packets, the act of validating an Initial Token and sending a Retry Packet is more expensive than the response to a non-Initial packet with an unknown Connection ID: simply dropping it and/or sending a Stateless Reset.

Nevertheless, a Retry Offload in Active Mode might desire to shield servers from non-Initial packets that do not correspond to a previously admitted Initial Packet. This has a number of considerations.

* If a Retry Offload maintains no per-flow state, it cannot distinguish between valid and invalid non-Initial packets and MUST forward all non-Initial Packets to the server.

* For QUIC versions the Retry Offload does not support and are present on the Allow-List (or absent from the Deny-List), the Retry Offload cannot distinguish Initial Packets from other long headers and therefore MUST admit all long headers.

* If a Retry Offload keeps per-flow state, it can identify 4-tuples

that have been previously approved, admit non-Initial packets from those flows, and drop all others. However, dropping short headers will effectively break Address Migration and NAT Rebinding when in Active Mode, as post-migration packets will arrive with a previously unknown 4-tuple. This policy will also break connection attempts using any new QUIC versions that begin connections with a short header.

* If a Retry Offload is integrated with a QUIC-LB routable load balancer [I-D.ietf-quic-load-balancers], it can verify that the Destination Connection ID is routable, and only admit non-Initial packets with routable DCIDs. As the Connection ID encoding is invariant across QUIC versions, the Retry Offload can do this for all short headers.

Nothing in this section prevents Retry Offloads from making basic syntax correctness checks on packets with QUIC versions that it understands (e.g., enforcing the Initial Packet datagram size minimum in version 1).

3. No-Shared-State Retry Offload

The no-shared-state Retry Offload requires no coordination, except that the server must be configured to accept this offload and know which QUIC versions the Retry Offload supports. The scheme uses the first bit of the token to distinguish between tokens from Retry packets (codepoint '0') and tokens from NEW_TOKEN frames (codepoint '1').

3.1. Configuration Agent Actions

See Section 2.

3.2. Offload Requirements

A no-shared-state Retry Offload MUST be present on all paths from potential clients to the server. These paths MUST fail to pass QUIC traffic should the offload fail for any reason. That is, if the offload is not operational, the server MUST NOT be exposed to client traffic. Otherwise, servers that have already disabled their Retry capability would be vulnerable to attack.

The path between offload and server MUST be free of any potential attackers. Note that this and other requirements above severely restrict the operational conditions in which a no-shared-state Retry Offload can safely operate.

Retry tokens generated by the offload MUST have the format below.

```
No-Shared-State Retry Offload Token {
  Token Type (1) = 0,
  ODCIL (7) = 8..20,
  Original Destination Connection ID (64..160),
  Opaque Data (..),
}
```

Figure 2: Format of non-shared-state Retry Offload tokens

The first bit of retry tokens generated by the offload MUST be zero. The token has the following additional fields:

ODCIL: The length of the original destination connection ID from the triggering Initial packet. This is in cleartext to be readable for the server, but authenticated later in the token. The Retry Offload SHOULD reject any token in which the value is less than 8.

Original Destination Connection ID: This also in cleartext and authenticated later.

Opaque Data: This data contains the information necessary to authenticate the Retry token in accordance with the QUIC specification. A straightforward implementation would encode the Retry Source Connection ID, client IP address, and a timestamp in the Opaque Data. A more space-efficient implementation would use the Retry Source Connection ID and Client IP as associated data in an encryption operation, and encode only the timestamp and the authentication tag in the Opaque Data. If the Initial packet alters the Connection ID or source IP address, authentication of the token will fail.

Upon receipt of an Initial packet with a token that begins with '0', the Retry Offload MUST validate the token in accordance with the QUIC specification.

In active mode, the offload MUST issue Retry packets for all client Initial packets that contain no token, or a token that has the first bit set to '1'. It MUST NOT forward the packet to the server. The offload MUST validate all tokens with the first bit set to '0'. If successful, the offload MUST forward the packet with the token intact. If unsuccessful, it MUST drop the packet. The Retry Offload MAY send an Initial Packet containing a CONNECTION_CLOSE frame with the INVALID_TOKEN error code when dropping the packet.

Note that this scheme has a performance drawback. When the Retry Offload is in active mode, clients with a token from a NEW_TOKEN frame will suffer a 1-RTT penalty even though its token provides proof of address.

In inactive mode, the offload MUST forward all packets that have no token or a token with the first bit set to '1'. It MUST validate all tokens with the first bit set to '0'. If successful, the offload MUST forward the packet with the token intact. If unsuccessful, it MUST drop the packet.

3.3. Server Requirements

A server behind a non-shared-state Retry Offload MUST NOT send Retry packets for a QUIC version the Retry Offload understands. It MAY send Retry for QUIC versions the Retry Offload does not understand.

Tokens sent in NEW_TOKEN frames MUST have the first bit set to '1'.

If a server receives an Initial Packet with the first bit in the token set to '1', it could be from a server-generated NEW_TOKEN frame and should be processed in accordance with the QUIC specification. If a server receives an Initial Packet with the first bit to '0', it is a Retry token and the server MUST NOT attempt to validate it. Instead, it MUST assume the address is validated, MUST include the packet's Destination Connection ID in a Retry Source Connection ID transport parameter, and MUST extract the Original Destination Connection ID from the token cleartext for use in the transport parameter of the same name.

4. Shared-State Retry Offload

A shared-state Retry Offload uses a shared key, so that the server can decode the offload's retry tokens. It does not require that all traffic pass through the Retry Offload, so servers MAY send Retry packets in response to Initial packets without a valid token.

Both server and offload MUST have time synchronized within two seconds of each other to prevent tokens being incorrectly marked as

expired.

The tokens are protected using AES128-GCM AEAD, as explained in Section 4.1. All tokens, generated by either the server or Retry Offload, MUST use the following format, which includes:

* A 1 bit token type identifier.

* A 7 bit token key identifier.

* A 96 bit unique token number transmitted in clear text, but protected as part of the AEAD associated data.

* A token body, encoding the Original Destination Connection ID and the Timestamp, optionally followed by server specific Opaque Data.

The token protection uses an 128 bit representation of the source IP address from the triggering Initial packet. The client IP address is 16 octets. If an IPv4 address, the last 12 octets are zeroes. It also uses the Source Connection ID of the Retry packet, which will cause an authentication failure if it differs from the Destination Connection ID of the packet bearing the token.

If there is a Network Address Translator (NAT) in the server infrastructure that changes the client IP, the Retry Offload MUST either be positioned behind the NAT, or the NAT must have the token key to rewrite the Retry token accordingly. Note also that a host that obtains a token through a NAT and then attempts to connect over a path that does not have an identically configured NAT will fail address validation.

The 96 bit unique token number is set to a random value using a cryptography-grade random number generator.

The token key identifier and the corresponding AEAD key and AEAD IV are provisioned by the configuration agent.

The token body is encoded as follows:

```
Shared-State Retry Offload Token Body {
   Timestamp (64),
   [ODCIL (8) = 8..20],
```

```
   [Original Destination Connection ID (64..160)],
   [Port (16)],
   Opaque Data (..),
}
```

Figure 3: Body of shared-state Retry Offload tokens

The token body has the following fields:

Timestamp: The Timestamp is a 64-bit integer, in network order, that expresses the expiration time of the token as a number of seconds in POSIX time (see Sec. 4.16 of [TIME_T]).

ODCIL: The original destination connection ID length. Tokens in NEW_TOKEN frames do not have this field.

Original Destination Connection ID: The server or Retry Offload copies this from the field in the client Initial packet. Tokens in NEW_TOKEN frames do not have this field.

Port: The Source Port of the UDP datagram that triggered the Retry packet. This field MUST be present if and only if the ODCIL is greater than zero. This field is therefore always absent in tokens in NEW_TOKEN frames.

Opaque Data: The server may use this field to encode additional information, such as congestion window, RTT, or MTU. The Retry Offload MUST have zero-length opaque data.

Some implementations of QUIC encode in the token the Initial Packet Number used by the client, in order to verify that the client sends the retried Initial with a PN larger that the triggering Initial. Such implementations will encode the Initial Packet Number as part of the opaque data. As tokens may be generated by the Service, servers MUST NOT reject tokens because they lack opaque data and therefore the packet number.

Shared-state Retry Offloads use the AES-128-ECB cipher. Future standards could add new algorithms that use other ciphers to provide cryptographic agility in accordance with [RFC7696]. Retry Offload and server implementations SHOULD be extensible to support new

algorithms.

4.1. Token Protection with AEAD

On the wire, the token is presented as:

```
Shared-State Retry Offload Token {
  Token Type (1),
  Key Sequence (7),
  Unique Token Number (96),
  Encrypted Shared-State Retry Offload Token Body (64..),
  AEAD Integrity Check Value (128),
}
```

Figure 4: Wire image of shared-state Retry Offload tokens

The tokens are protected using AES128-GCM as follows:

* The Key Sequence is the 7 bit identifier to retrieve the token key and IV.

* The AEAD IV, is 96 bits generated by the configuration agent.

* The AEAD nonce, N, is formed by XORing the AEAD IV with the 96 bit unique token number.

* The associated data is a formatted as a pseudo header by combining the cleartext part of the token with the IP address of the client. The format of the pseudoheader depends on whether the Token Type bit is '1' (a NEW_TOKEN token) or '0' (a Retry token).

```
Shared-State Retry Offload Token Pseudoheader {
  IP Address (128),
  Token Type (1),
  Key Sequence (7),
  Unique Token Number (96),
  [RSCIL (8)],
  [Retry Source Connection ID (0..20)],
}
```

Figure 5: Psuedoheader for shared-state Retry Offload tokens

RSCIL: The Retry Source Connection ID Length in octets. This field is only present when the Token Type is '0'.

Retry Source Connection ID: To create a Retry Token, populate this field with the Source Connection ID the Retry packet will use. To validate a Retry token, populate it with the Destination Connection ID of the Initial packet that carries the token. This field is only present when the Token Type is '0'.

* The input plaintext for the AEAD is the token body. The output ciphertext of the AEAD is transmitted in place of the token body.

* The AEAD Integrity Check Value(ICV), defined in Section 6 of [RFC4106], is computed as part of the AEAD encryption process, and is verified during decryption.

4.2. Configuration Agent Actions

The configuration agent generates and distributes a "token key", a "token IV", a key sequence, and the information described in Section 2.

4.3. Offload Requirements

In inactive mode, the Retry Offload forwards all packets without further inspection or processing. The rest of this section only applies to a offload in active mode.

Retry Offloads MUST NOT issue Retry packets except where explicitly allowed below, to avoid sending a Retry packet in response to a Retry token.

The offload MUST generate Retry tokens with the format described above when it receives a client Initial packet with no token.

If there is a token of either type, the offload MUST attempt to decrypt it.

To decrypt a packet, the offload checks the Token Type and constructs

a pseudoheader with the appropriate format for that type, using the bearing packet's Destination Connection ID to populate the Retry Source Connection ID field, if any.

A token is invalid if:

* it uses an unknown key sequence,

* the AEAD ICV does not match the expected value (By construction, it will only match if the client IP Address, and any Retry Source Connection ID, also matches),

* the ODCIL, if present, is invalid for a client-generated CID (less than 8 or more than 20 in QUIC version 1),

* the Timestamp of a token points to time in the past (however, in order to allow for clock skew, it SHOULD NOT consider tokens to be expired if the Timestamp encodes less than two seconds in the past), or

* the port number, if present, does not match the source port in the encapsulating UDP header.

Packets with valid tokens MUST be forwarded to the server.

The offload MUST drop packets with invalid tokens. If the token is of type '1' (NEW_TOKEN), it MUST respond with a Retry packet. If of type '0', it MUST NOT respond with a Retry packet.

4.4. Server Requirements

The server MAY issue Retry or NEW_TOKEN tokens in accordance with [RFC9000]. When doing so, it MUST follow the format above.

The server MUST validate all tokens that arrive in Initial packets, as they may have bypassed the Retry Offload. It determines validity using the procedure in Section 4.3.

If a valid Retry token, the server populates the original_destination_connection_id transport parameter using the corresponding token field. It populates the retry_source_connection_id transport parameter with the Destination

Connection ID of the packet bearing the token.

In all other respects, the server processes both valid and invalid tokens in accordance with [RFC9000].

For QUIC versions the offload does not support, the server MAY use any token format.

5. Security Considerations

5.1. Shared-State Retry Keys

The Shared-State Retry Offload defined in Section 4 describes the format of retry tokens or new tokens protected and encrypted using AES128-GCM. Each token includes a 96 bit randomly generated unique token number, and an 8 bit identifier used to get the AES-GCM encryption context. The AES-GCM encryption context contains a 128 bit key and an AEAD IV. There are three important security considerations for these tokens:

* An attacker that obtains a copy of the encryption key will be able to decrypt and forge tokens.

* Attackers may be able to retrieve the key if they capture a sufficently large number of retry tokens encrypted with a given key.

* Confidentiality of the token data will fail if separate tokens reuse the same 96 bit unique token number and the same key.

To protect against disclosure of keys to attackers, offload and servers MUST ensure that the keys are stored securely. To limit the consequences of potential exposures, the lifetime of any given key should be limited.

Section 6.6 of [RFC9001] states that "Endpoints MUST count the number of encrypted packets for each set of keys. If the total number of encrypted packets with the same key exceeds the confidentiality limit for the selected AEAD, the endpoint MUST stop using those keys." It goes on with the specific limit: "For AEAD_AES_128_GCM and AEAD_AES_256_GCM, the confidentiality limit is 2^23 encrypted packets; see Appendix B.1." It is prudent to adopt the same limit here, and configure the offload in such a way that no more than 2^23 tokens are generated with the same key.

In order to protect against collisions, the 96 bit unique token numbers should be generated using a cryptographically secure pseudorandom number generator (CSPRNG), as specified in Appendix C.1 of the TLS 1.3 specification [RFC8446]. With proper random numbers, if fewer than 2^40 tokens are generated with a single key, the risk of collisions is lower than 0.001%.

6. IANA Considerations

There are no IANA requirements.

7. References

7.1. Normative References

[RFC8446] Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", RFC 8446, DOI 10.17487/RFC8446, August 2018, <https://www.rfc-editor.org/info/rfc8446>.

[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/info/rfc9000>.

[TIME_T] "Open Group Standard: Vol. 1: Base Definitions, Issue 7", IEEE Std 1003.1 , 2018, <http://pubs.opengroup.org/onlinepubs/9699919799/basedefs/V1_chap04.html#tag_04_16>.

7.2. Informative References

[I-D.ietf-quic-load-balancers] Duke, M., Banks, N., and C. Huitema, "QUIC-LB: Generating Routable QUIC Connection IDs", Work in Progress, Internet-Draft, draft-ietf-quic-load-balancers-13, 28 March 2022, <https://www.ietf.org/archive/id/draft-ietf-quic-load-balancers-13.txt>.

[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>.

[RFC4106] Viega, J. and D. McGrew, "The Use of Galois/Counter Mode (GCM) in IPsec Encapsulating Security Payload (ESP)", RFC 4106, DOI 10.17487/RFC4106, June 2005, <https://www.rfc-editor.org/info/rfc4106>.

[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>.

[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/info/rfc6020>.

[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/info/rfc7696>.

[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/info/rfc8340>.

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

[RFC9001] Thomson, M., Ed. and S. Turner, Ed., "Using TLS to Secure QUIC", RFC 9001, DOI 10.17487/RFC9001, May 2021, <https://www.rfc-editor.org/info/rfc9001>.

Appendix A. Retry Offload YANG Model

These YANG models conform to [RFC6020] and express a complete Retry Offload configuration.

```
module ietf-retry-offload {
  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 offloads that generate Retry packets on their behalf.

       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 "2022-02-11" {
      description
        "Initial version";
      reference
        "RFC XXXX, QUIC Retry Offloads";
    }

    container retry-offload-config {
      description
        "Configuration of Retry Offload. If supported-versions is empty,
         there is no Retry Offload. If token-keys is empty, it uses the
         non-shared-state offload. If present, it uses shared-state
         tokens.";
```



```
      leaf-list supported-versions {
        type uint32;
        description
          "QUIC versions that the Retry Offload supports. If empty,
           there is no Retry Offload.";
      }

      leaf unsupported-version-default {
        type enumeration {
          enum allow {
            description "Unsupported versions admitted by default";
          }
          enum deny {
            description "Unsupported versions denied by default";
          }
        }
        default allow;
        description
          "Are unsupported versions not in version-exceptions allowed
           or denied?";
      }

      leaf-list version-exceptions {
        type uint32;
        description
          "Exceptions to the default-deny or default-allow rule.";
```

```
      }

      list token-keys {
        key "key-sequence-number";
        description
          "list of active keys, for key rotation purposes. Existence
           implies shared-state format";

        leaf key-sequence-number {
          type uint8 {
            range "0..127";
          }
          mandatory true;
          description
            "Identifies the key used to encrypt the token";
          }

        leaf token-key {
          type retry-offload-key;
          mandatory true;
          description
            "16-byte key to encrypt the token";
```



```
        }

        leaf token-iv {
          type yang:hex-string {
            length 23;
          }
          mandatory true;
          description
            "8-byte IV to encrypt the token, encoded in 23 bytes";
        }
      }
    }
  }
```

A.1. Tree Diagram

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

```
module: retry-offload-config
```

```
  +--rw retry-offload-config
     +--rw supported-versions*            uint32
     +--rw unsupported-version-default?   enumeration
     +--rw version-exceptions*            uint32
     +--rw token-keys* [key-sequence-number]
        +--rw key-sequence-number    uint8
        +--rw token-key              quic-lb-key
        +--rw token-iv               yang:hex-string
```

Shared State Retry Token Test Vectors

In this case, the shared-state retry token is issued by Retry Offload, so the opaque data of shared-state retry token body would be null ({{shared-state-retry}}).

Configuration: key_seq 0x00 encrypt_key 0x303132333435363738393031323334353 AEAD_IV 0x313233343536373839303132

Shared-State Retry Offload Token Body: ODCIL 0x12 RSCIL 0x10 port 0x1a0a original_destination_connection_id 0x0c3817b544ca1c94313bba41757547eec937 retry_source_connection_id 0x0301e770d24b3b13070dd5c2a9264307 timestamp 0x0000000060c7bf4d

Shared-State Retry Offload Token: unique_token_number 0x59ef316b70575e793e1a8782 key_sequence 0x00 encrypted_shared_state_retry_offload_token_body 0x7d38b274aa4427c7a1557c3fa666945931defc65da387a83855196a7cb73caac1e28e5346fd76868de94f8b62294 AEAD_ICV 0xf91174fdd711543a32d5e959867f9c22

AEAD related parameters: client_ip_addr 127.0.0.1 client_port 6666 AEAD_nonce 0x68dd025f45616941072ab6b0 AEAD_associated_data 0x7f0000010000000000000000000000059ef316b70575e793e1a878200 ~~~

Appendix B. Transition Mode Scenarios

The logic motivating transition mode behavior involves detailed reasoning about endpoint behavior during the handshake. This non-normative appendix walks through the scenarios.

Dropping Initial packets in the client's second flight can cause performance problems or deadlocks. In the case where the client and

server first flight end with both sides having handshake keys, there will generally be no impact on performance. However, if an Initial ACK is critical to progress, as it can be in the case of multiple-packet TLS messages, Hello Retry Requests, and similar cases, dropping subsequent Initial ACKs results in deadlock.

In transition mode, the Retry Offload forwards Initials with no token while also generating a Retry. This allows handshakes to progress without further incident.

B.1. Handshakes in Progress

If the client hello was admitted in inactive mode, then the client has already received a packet from the server. Although subsequent client Initial packets will trigger a Retry, the client will ignore these packets. Those Initials will also be processed by the server to continue the handshake.

B.2. New Connections

After sending a Client Hello in Initial Packet A, a client will rapidly receive a Retry Packet from the Offload and attempt to reconnect accordingly with Initial Packet B.

The client will discard any server response to Initial A. If a Retry, it is a second Retry on the connection. If an Initial, its is encrypted with keys derived from Initial A, which have already been discarded, and will be a decryption failure.

Initial B's destination connection ID will be new, so the server will process it as a new connection and proceed normally.

Unfortunately, the server connection state initiated by Initial A will remain. For this reason, this document suggests that servers silently terminate the older connection. Requiring the address to be validated avoids cases where an attacker simply replays a client Initial with a new Destination Connection ID to terminate a valid

connection.

Note that there are corner cases involving further packet loss that result in connection timeout. For instance, if the Retry Offload's response to Initial A is lost, then the connection will proceed based on Initial A. If the Retry Offload then switches from transition mode to active mode before the client's second flight arrives, the Retry Offload will drop the Initial packet in that flight, and the connection might deadlock.

Appendix C. Acknowledgments

Christian Huitema, Ling Tao Nju, and William Zeng Ke all provided useful input to this document.

Appendix D. Change Log

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

D.1. since draft-duke-quic-retry-offload-00

* Converted to adopted IETF draft
* Cleaner transition from inactive to active mode

D.2. since draft-ietf-quic-load-balancers-12

* Separated from the QUIC-LB draft
* Renamed "Retry Service" to "Retry Offload"

D.3. since draft-ietf-quic-load-balancers-11

* Fixed mistakes in test vectors

D.4. 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

D.5. 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.

D.6. since draft-ietf-quic-load-balancers-08

* Eliminate Dynamic SID allocation

* Eliminated server use bytes

D.7. 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

D.8. 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 Offload considerations for non-Initial packets

D.9. 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

D.10. 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

D.11. since-draft-ietf-quic-load-balancers-03

* Improved Config Rotation text
* Added stream cipher test vectors
* Deleted the Obfuscated CID algorithm

D.12. 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

D.13. since-draft-ietf-quic-load-balancers-01

* Test vectors for load balancer decoding

* Deleted remnants of in-band protocol

* Light edit of Retry Offloads section

* Discussed load balancer chains

D.14. since-draft-ietf-quic-load-balancers-00

* Removed in-band protocol from the document

D.15. Since draft-duke-quic-load-balancers-06

* Switch to IETF WG draft.

D.16. 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

D.17. Since draft-duke-quic-load-balancers-04

* Added standard for Retry Offloads

D.18. 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

D.19. 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

D.20. Since draft-duke-quic-load-balancers-01

* Complete rewrite

* Supports multiple security levels

* Lightweight messages

D.21. 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