tls B. Schwartz
Internet-Draft Google LLC
Intended status: Standards Track July 02, 2019
Expires: January 3, 2020

# TLS Metadata for Load Balancers
## draft-schwartz-tls-lb-01

Abstract

A load balancer that does not terminate TLS may wish to provide some information to the backend server, in addition to forwarding TLS data. This draft proposes a protocol between load balancers and backends that enables secure, efficient delivery of TLS with additional information. The need for such a protocol has recently become apparent in the context of split mode ESNI.

Table of Contents



## 1. Conventions and Definitions

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

Data encodings are expressed in the TLS 1.3 presentation language, as defined in Section 3 of [TLS13].

## 2. Background

A load balancer is a server or bank of servers that acts as an intermediary between the client and a range of backend servers. As the name suggests, a load balancer's primary function is to ensure that client traffic is spread evenly across the available backend servers. However load balancers also serve many other functions, such as identifying connections intended for different backends and forwarding them appropriately, or dropping connections that are deemed malicious.

A load balancer operates at a specific point in the protocol stack, forwarding e.g. IP packets, TCP streams, TLS contents, HTTP requests, etc. Most relevant to this proposal are TCP and TLS load balancers. TCP load balancers terminate the TCP connection with the client and establish a new TCP connection to the selected backend, bidirectionally copying the TCP contents between these two connections. TLS load balancers additionally terminate the TLS connection, forwarding the plaintext to the backend server (typically inside a new TLS connection). TLS load balancers must therefore hold the private keys for the domains they serve.

When a TCP load balancer forwards a TLS stream, the load balancer has no way to incorporate additional information into the stream. Insertion of any additional data would cause the connection to fail. However, the load-balancer and backend can share additional information if they agree to speak a new protocol. The most popular protocol used for this purpose is currently the PROXY protocol [PROXY], developed by HAPROXY. This protocol prepends a plaintext collection of metadata (e.g. client IP address) onto the TCP socket. The backend can parse this metadata, then pass the remainder of the stream to its TLS library.

The PROXY protocol is widely used, but it offers no confidentiality or integrity protection, and therefore might not be suitable when the load balancer and backend communicate over the public internet.

## 3. Goals

- Enable TCP load balancers to forward metadata to the backend.
- Reduce the need for TLS-terminating load balancers.
- Ensure confidentiality and integrity for all forwarded metadata.
- Enable split ESNI architectures.
- Prove to the backend that the load balancer intended to associate this metadata with this connection.
- Achieve good CPU and memory efficiency.
- Don't impose additional latency.
- Support backends that receive a mixture of direct and load-balanced TLS.
- Support use in QUIC.

o Enable simple and safe implementation.

## 4. Overview

The proposed protocol provides one-way communication from a load balancer to a backend server. It works by prepending information to the forwarded connection:

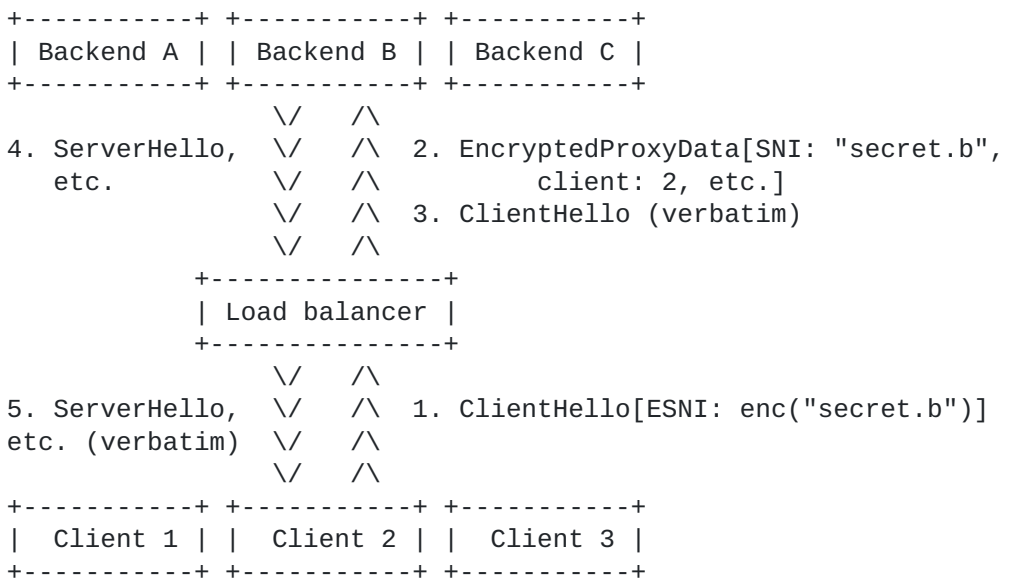


Figure 1: Data flow diagram

## 5. Encoding

A ProxyExtension is identical in form to a standard TLS Extension (Section 4.2 of [TLS13]), with a new identifier space for the extension types.

```
struct {
  ProxyExtensionType extension_type;
  opaque extension_data<0..2^16-1>;
} ProxyExtension;
```

The ProxyData contains a set of ProxyExtensions.

```
struct {
  ProxyExtension proxy_data<0..2^16-1>;
} ProxyData;
```

The EncryptedProxyData structure contains metadata associated with the original ClientHello (Section 4.1.2 of [TLS13]), encrypted with a pre-shared key that is configured out of band.

```
struct {
  opaque psk_identity<1..2^16-1>;
  opaque nonce<8..2^16-1>
  opaque encrypted_proxy_data<1..2^16-1>;
} EncryptedProxyData;
```

- psk_identity: The identity of a PSK previously agreed upon by the load balancer and the backend. Including the PSK identity allows for updating the PSK without disruption.

- nonce: Non-repeating initializer for the AEAD. This prevents an attacker from observing whether the same ClientHello is marked with different metadata over time.

- encrypted_proxy_data: AEAD-Encrypt(key, nonce, additional_data=ClientHello, plaintext=ProxyData). The key and AEAD function are agreed out of band and associated with psk_identity.

When the load balancer receives a ClientHello, it serializes any relevant metadata into a ProxyData, then encrypts it with the ClientHello as additional data, to produce EncryptedProxyData.

## 6. Defined ProxyExtensions

Like a standard TLS Extension, a ProxyExtension is identified by a 2-byte type number. There are initially three type numbers allocated:

```
enum {
  padding(0),
  network_address(1),
  esni_inner(2),
  (65535)
} ProxyExtensionType;
```

The "padding" extension functions as described in [RFC7685]. It is used here to avoid leaking information about the other extensions.

The "network_address" extension functions as described in [I-D.kinnear-tls-client-net-address]. It conveys the client IP address observed by the load balancer.

The "esni_inner" extension can only be used if the ClientHello contains the encrypted_server_name extension [ESNI]. The extension_data is the ClientESNIInner (Section 5.1.1 of [ESNI]), which contains the true SNI and nonce. This is useful when the load balancer knows the ESNI private key and the backend does not, i.e. split mode ESNI.

Load balancers SHOULD only include extensions that are specified for use in ProxyData, and backends MUST ignore any extensions that they do not recognize.

## 7. Use with TLS over TCP

When forwarding a TLS stream over TCP, the load balancer SHOULD send a ProxyHeader at the beginning of the stream:

```
struct {
  uint8 opaque_type = 0;
  ProtocolVersion version = 0;
  uint16 length = length(ProxyHeader.contents);
  EncryptedProxyData contents;
} ProxyHeader;
```

The opaque_type field ensures that this header is distinguishable from an ordinary TLS connection, whose first byte is always 22 (ContentType = handshake in Section 5.1 of [TLS13]). This structure matches the layout of TLSPlaintext with a ContentType of "invalid", potentially simplifying parsing.

Following the ProxyHeader, the load balancer MUST send the full contents of the TCP stream, exactly as received from the client. The backend will observe the ProxyHeader, immediately followed by a TLSPlaintext frame containing the ClientHello. The backend will decrypt the ProxyHeader using the ClientHello as associated data, and process the ClientHello and the remainder of the stream as standard TLS.

When receiving a ProxyHeader with an unrecognized version, the backend SHOULD ignore this ProxyHeader and proceed as if the following byte were the first byte received.

## 8. Use with QUIC

A QUIC load balancer provides this service by extracting the ClientHello from any client Initial packet [I-D.ietf-quic-tls]. A multi-tenant load balancer needs to perform this extraction anyway in order to determine where the connection should be forwarded, either by SNI or ESNI.

Extracting a TLS ClientHello from a QUIC handshake is a version-dependent action, so a load balancer cannot support unrecognized versions of QUIC. If the load balancer receives a packet with an unrecognized QUIC version, it MUST reply with a VersionNegotiation packet indicating the supported versions (currently only version 1). If the backend applies downgrade protection, it SHOULD account for the impact of the load balancer.

In QUIC version 1, each handshake begins with an Initial packet sent by the client. This packet uses the QUIC "long header" packet form, starting with a "fixed bit" of 1 and a "frame type" of 0x0.

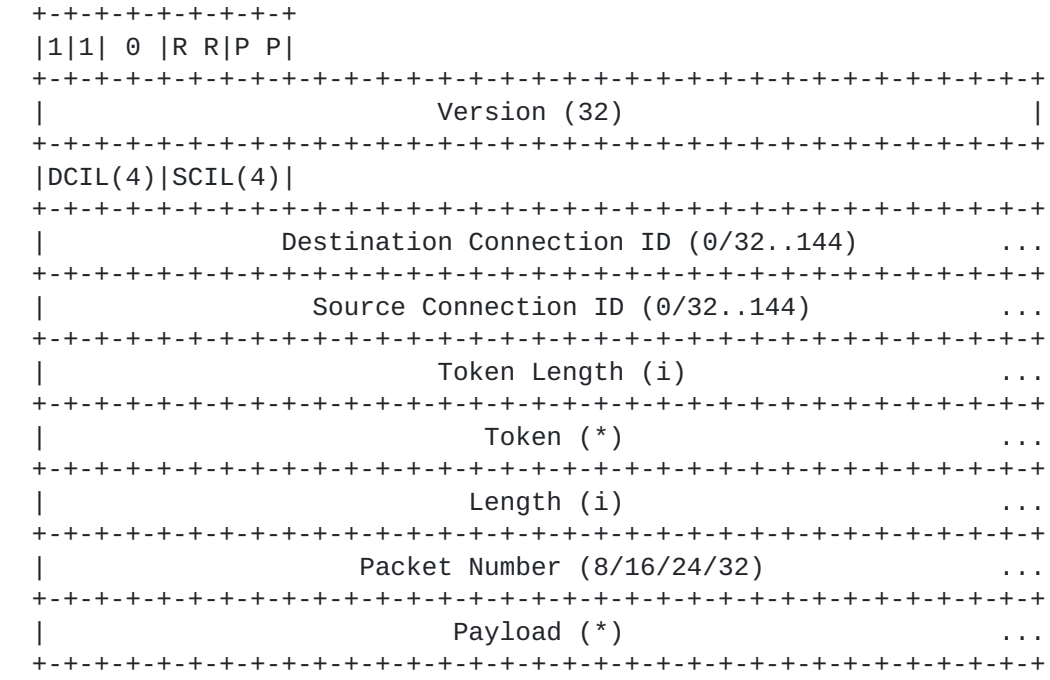


Figure 2: QUIC Initial Packet

A client Initial packet contains a complete ClientHello, in a CRYPTO frame in the payload. The load balancer extracts this ClientHello in order to compute EncryptedProxyData.

TODO: Confirm that HelloRetryRequest elicits an Initial containing a complete ClientHello. The QUIC draft text is unclear.

To send EncryptedProxyData to the backend, the load balancer constructs a new packet with a header copied from the Initial, but with a frame type of 0x1 and a new version (0xTBD). Its payload consists of the old Initial's version number (currently always 1) and the EncryptedProxyData.

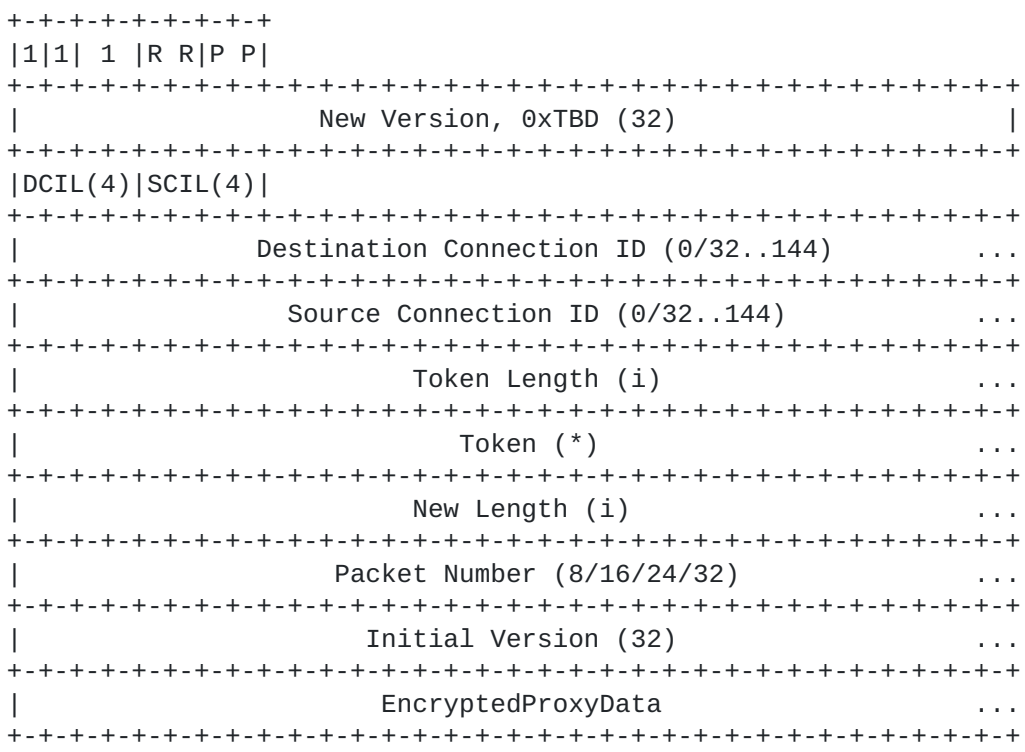


Figure 3: EncryptedProxyData packet to the backend

The load balancer then forwards the client Initial unmodified, except for replacing its Version number with 0xTBD. All other QUIC packets are forwarded entirely unmodified.

The backend, upon receipt of a packet with QUIC version 0xTBD and type "0" or "1", waits for a second packet with version 0xTBD, the other type value, and matching connection IDs, token, and packet number. When both packets have been received, the backend can reconstruct the original Initial packet and decrypt the EncryptedProxyData.

If the second packet is not received within a brief time period (e.g. 100 ms), the backend SHOULD discard the first packet.

## 9. Configuration

The method of configuring of the PSK on the load balancer and backend is not specified here. However, the PSK MAY be represented as a ProxyKey:

```
   struct {
     ProtocolVersion version = 0;
     opaque psk_identity<1..2^16-1>;
     CipherSuite cipher_suite;
     opaque key<16..2^16-1>
   } ProxyKey;
```

## 10. Security considerations

### 10.1. Integrity

This protocol is intended to provide the backend with a strong guarantee of integrity for the metadata written by the load balancer.

For example, an active attacker cannot take metadata intended for one stream and attach it to another, because each stream will have a unique ClientHello, and the metadata is bound to the ClientHello by AEAD.

One exception to this protection is in the case of an attacker who deliberately reissues identical ClientHello messages. An attacker who reuses a ClientHello can also reuse the metadata associated with it, if they can first observe the EncryptedProxyData transferred between the load balancer and the backend. This could be used by an attacker to reissue data originally generated by a true client (e.g. as part of a 0-RTT replay attack), or it could be used by a group of adversaries who are willing to share a single set of client secrets while initiating different sessions, in order to reuse metadata that they find helpful.

As such, the backend SHOULD treat this metadata as advisory.

### 10.2. Confidentiality

This protocol is intended to maintain confidentiality of the metadata transferred between the load balancer and backend, currently consisting of the ESNI plaintext and the client IP address. An observer between the client and the load balancer does not observe this protocol at all, and an observer between the load balancer and backend observes only ciphertext.

However, an adversary who can monitor both of these links can easily observe that a connection from the client to the load balancer is shortly followed by a connection from the load balancer to a backend, with the same ClientHello. This reveals which backend server the client intended to visit. In many cases, the choice of backend server could be the sensitive information that ESNI is intended to protect.

## 11. IANA Considerations

Need to create a new ProxyExtensionType registry.

Need to allocate TBD as a reserved QUIC version code.

## 12. References

### 12.1. Normative References

[ESNI] Rescorla, E., Oku, K., Sullivan, N., and C. Wood, "Encrypted Server Name Indication for TLS 1.3", draft-ietf-tls-esni-03 (work in progress), March 2019.

[I-D.ietf-quic-tls] Thomson, M. and S. Turner, "Using TLS to Secure QUIC", draft-ietf-quic-tls-20 (work in progress), April 2019.

[I-D.kinnear-tls-client-net-address] Kinnear, E., Pauly, T., and C. Wood, "TLS Client Network Address Extension", draft-kinnear-tls-client-net-address-00 (work in progress), March 2019.

[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>.

[RFC7685] Langley, A., "A Transport Layer Security (TLS) ClientHello Padding Extension", RFC 7685, DOI 10.17487/RFC7685, October 2015, <https://www.rfc-editor.org/info/rfc7685>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/info/rfc8174>.

[TLS13] 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>.

### 12.2. Informative References

[PROXY] Tarreau, W., "The PROXY protocol", March 2017, <https://www.haproxy.org/download/1.8/doc/proxy-protocol.txt>.

## Appendix A. Acknowledgements

This is an elaboration of an idea proposed by Eric Rescorla during the development of ESNI. Thanks to David Schinazi and David Benjamin for suggesting important improvements.

## Appendix B. Open Questions

Should the ProxyExtensionType registry have a reserved range for private extensions?

## Author's Address

Benjamin M. Schwartz
Google LLC

Email: bemasc@google.com