tls B.
Schwartz
Internet-Draft Google
LLC
Intended status: Standards Track October 31,
2019
Expires: May 3, 2020

TLS Metadata for Load Balancers
draft-schwartz-tls-lb-02

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.

Status of This Memo

Copyright Notice

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 effective and 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. It also does not offer a way for the backend to reply.

3. Goals

- o Enable TCP load balancers to forward metadata to the backend.

- o Enable backends to reply.

- o Reduce the need for TLS-terminating load balancers.

- o Ensure confidentiality and integrity for all forwarded metadata.

- o Enable split ESNI architectures.

- o Prove to the backend that the load balancer intended to associate this metadata with this connection.

- o Achieve good CPU and memory efficiency.

- Don't impose additional latency.

- Support backends that receive a mixture of direct and load-balanced TLS.

- Enable simple and safe implementation.

4. Overview

The proposed protocol supports a two-way exchange between a load balancer and a backend server. It works by prepending information to the TLS handshake:

```
        +-----------+ +-----------+ +-----------+
        | Backend A | | Backend B | | Backend C |
        +-----------+ +-----------+ +-----------+
                          \/   /\
   4. EncryptedProxyData[ \/   /\  3. ClientHello (verbatim)
       got SNI info]      \/   /\  2. EncryptedProxyData[
   5. ServerHello, etc.   \/   /\       SNI="secret.b",
                          \/   /\       client=2, etc.]
                          \/   /\
                     +---------------+
                     | Load balancer |
                     +---------------+
                          \/   /\
   6. ServerHello, etc.   \/   /\  1. ClientHello[
      (verbatim)          \/   /\       ESNI=enc("secret.b")]
                          \/   /\
         +-----------+ +-----------+ +-----------+
         |  Client 1 | |  Client 2 | |  Client 3 |
         +-----------+ +-----------+ +-----------+
```

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;
```

ProxyExtensions can be sent in an upstream (to the backend) or downstream (to the load balancer) direction

```
   enum {
     upstream(0),
     downstream(1),
     (255)
   } ProxyDataDirection;
```

The ProxyData contains a set of ProxyExtensions.

```
   struct {
     ProxyDataDirection direction;
     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, plaintext=ProxyData)". The key and AEAD function are agreed out of band and associated with "psk_identity". The "additional_data" is context-dependent.

When the load balancer receives a ClientHello, it serializes any relevant metadata into an upstream ProxyData, then encrypts it with the ClientHello as "additional_data" to produce the EncryptedProxyData. The backend's reply is a downstream ProxyData struct, also transmitted as an EncryptedProxyData, using the upstream EncryptedProxyData as "additional_data". Recipients in each case MUST verify that "ProxyData.direction" has the expected value, and discard the connection if it does not.

The downstream ProxyData SHOULD NOT contain any ProxyExtensionType values that were not present in the upstream ProxyData.

6. Defined ProxyExtensions

Like a standard TLS Extension, a ProxyExtension is identified by a uint16 type number. Load balancers MUST only include extensions that are registered for use in ProxyData. Backends MUST ignore any extensions that they do not recognize.

There are initially seven type numbers allocated:

```
enum {
  padding(0),
  client_address(1),
  destination_address(2),
  esni_inner(3),
  certificate_padding(4),
  overload(5),
  ratchet(6),
  (65535)
} ProxyExtensionType;
```

6.1. padding

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

6.2. client_address

The "client_address" extension functions as described in [I-D.kinnear-tls-client-net-address]. It conveys the client IP address observed by the load balancer. Backends that make use of this extension SHOULD include an empty "client_address" extension in the downstream ProxyData.

6.3. destination_address

The "destination_address" extension is identical to the "client_address" extension, except that it contains the load balancer's server IP address that received this connection.

6.4. esni_inner

The "esni_inner" extension is only sent upstream, and 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.

6.5. certificate_padding

The "certificate_padding" extension always contains a single uint32 value. The upstream value conveys the padding granularity "G", and the downstream value indicates the unpadded size of the Certificate struct (Section 4.4.2 of [TLS13]).

To pad the Handshake message (Section 4 of [TLS13]) containing the Certificate struct, the backend SHOULD select the smallest "length_of_padding" (Section 5.2 of [TLS13]) such that "Handshake.length + length_of_padding" is a multiple of "G".

The load balancer SHOULD include this extension whenever it sends the "esni_inner" extension.

Padding certificates from many backends to the same length is important to avoid revealing which backend is responding to a ClientHello. Load balancer operators SHOULD ensure that no backend has a unique certificate size after padding, and MAY set "G" large enough to make all responses have equal size.

6.6. overload

In the upstream ProxyData, the "overload" extension contains a single uint16 indicating the approximate proportion of connections that are being routed to this server as a fraction of 65535. If there is only one server, load balancers SHOULD set the value to 65535.

In the downstream ProxyData, the value is an OverloadValue:

```
enum {
  accepted(0),
  overloaded(1),
  rejected(2),
  (255)
} OverloadState;
struct {
  OverloadState state;
  uint16 load;
  uint32 ttl;
} OverloadValue;
```

When "OverloadValue.state" is "accepted", the backend is accepting connections normally. The "overloaded" state indicates that the backend is accepting this connection, but would prefer not to receive additional connections. A value of "rejected" indicates that the backend did not accept this connection. When sending a "rejected"

response, the backend SHOULD close the connection without sending a ServerHello.

"OverloadValue.load" indicates the load fraction of the responding backend server, with 65535 indicating maximum load.

The load balancer SHOULD treat this information as valid for "OverloadValue.ttl" seconds, or until it receives another OverloadValue from that server.

Load balancers that have multiple available backends for an origin SHOULD avoid connecting to servers that are in the "overloaded" or "rejected" state. When a connection is rejected, the load balancer MAY retry that connection by sending the ClientHello to a different backend server. When multiple servers are in the "accepted" state, the load balancer MAY use "OverloadValue.load" to choose among them.

When there is a server in an unknown state (i.e. a new server or one whose last TTL has expired), the load balancer SHOULD direct at least one connection to it, in order to refresh its OverloadState.

If all servers are in the "overloaded" or "rejected" state, the load balancer SHOULD drop the connection.

6.7. ratchet

If the backend server is reachable without traversing the load balancer, and an adversary can observe packets on the link between the load balancer and the backend, then that adversary can execute a replay flooding attack, sending the backend server duplicate copies of observed EncryptedProxyData and ClientHello. This attack can waste server resources on the Diffie-Hellman operations required to process the ClientHello, resulting in denial of service.

The "ratchet" extension reduces the impact of such an attack on the backend server by allowing the backend to reject these duplicates after decrypting the ProxyData. (This decryption uses only a symmetric cipher, so it is expected to be much faster than typical Diffie-Hellman operations.) Its upstream payload consists of a RatchetValue:

```
struct {
  uint64 index;
  uint64 floor;
} RatchetValue;
```

A RatchetValue is scoped to a single backend server and "psk_identity". Within that scope, the load balancer initializes "index" to a random value, and executes the following procedure:

1. For each new forwarded connection (to the same server under the same "psk_identity"), increment "index".

2. Set "floor" to the "index" of the earliest connection that has not yet been connected or closed.

The backend server initializes "floor" to the first "RatchetValue.floor" it receives (under a "psk_identity"), and then executes the following procedure for each incoming connection:

1. Define "a >= b" if the most significant bit of "a - b" is 0.

2. Let "newValue" be the RatchetValue in the ProxyData.

3. If "newValue.index < floor", ignore the connection.

4. If "newValue.floor >= floor", set "floor" to "newValue.floor".

5. OPTIONALLY, ignore the connection if "newValue.index" has been seen recently. This can be implemented efficiently by keeping track of any "index" values greater than "floor" that appear to have been skipped.

With these measures in place, replays can be rejected without processing the ClientHello.

In principle, this replay protection fails after 2^64 connections when the "floor" value wraps. On a backend server that averages 10^9 new connections per second, this would occur after 584 years. To avoid this replay attack, load balancers and backends SHOULD establish a new PSK at least this often.

Backends that are making use of the "ratchet" extension SHOULD include an empty "ratchet" extension in their downstream ProxyData.

7. Protocol wire format

When forwarding a TLS stream over TCP, the load balancer SHOULD prepend a TLSPlaintext whose "content_type" is XX (proxy_header) and whose "fragment" is the EncryptedProxyData.

Following this proxy header, the load balancer MUST send the full contents of the TCP stream, exactly as received from the client. The backend will observe the proxy header, immediately followed by a

TLSPlaintext containing the ClientHello. The backend will decrypt the EncryptedProxyData using the ClientHello as associated data, and process the ClientHello and the remainder of the stream as standard TLS.

Similarly, the backend SHOULD reply with the downstream EncryptedProxyData in a proxy header, followed by the normal TLS stream, beginning with a TLSPlaintext frame containing the ServerHello. If the downstream ProxyHeader is not present, has an unrecognized version number, or produces an error, the load balancer SHOULD proxy the rest of the stream regardless.

8. Security considerations

8.1. Integrity

This protocol is intended to provide both parties with a strong guarantee of integrity for the metadata they receive. 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.

Backends that are sensitive to this attack SHOULD implement the "ratchet" mechanism in Section 6.7, including the optional defenses.

8.2. Confidentiality

This protocol is intended to maintain confidentiality of the metadata transferred between the load balancer and backend, especially 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.

8.3. Fingerprinting

Connections to different domains might be distinguishable by the cleartext contents of the ServerHello, such as "cipher_suite" and "server_share.group". Load balancer operators with ESNI support SHOULD provide backend operators with a list of cipher suites and groups to support, and a preference order, to avoid different backends having distinctive behaviors.

9. IANA Considerations

IANA will be directed to add the following allocation to the TLS ContentType registry:

Value	Description	DTLS-OK	Reference
XX	proxy_header	N	This document

IANA will be directed to create a new "TLS ProxyExtensionType Values" registry on the TLS Extensions page. Values less than 0x8000 will be subject to the "RFC Required" registration procedure, and the rest will be "First Come First Served". To avoid codepoint exhaustion, proxy developers SHOULD pack all their nonstandard information into a single ProxyExtension.

10. References

10.1. Normative References

[ESNI] Rescorla, E., Oku, K., Sullivan, N., and C. Wood, "Encrypted Server Name Indication for TLS 1.3", draft-ietf-tls-esni-04 (work in progress), July 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>.

10.2. 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, David Benjamin, and Piotr Sikora for suggesting important improvements.

Author's Address

Benjamin M. Schwartz
Google LLC

Email: bemasc@google.com