TLS
Internet-Draft
Updates: 6347 (if approved)
Intended status: Standards Track
Expires: May 6, 2021

E. Rescorla, Ed.
RTFM, Inc.
H. Tschofenig, Ed.
T. Fossati
Arm Limited

November 02, 2020

Connection Identifiers for DTLS 1.2
draft-ietf-tls-dtls-connection-id-08

Abstract

This document specifies the Connection ID (CID) construct for the Datagram Transport Layer Security (DTLS) protocol version 1.2.

A CID is an identifier carried in the record layer header that gives the recipient additional information for selecting the appropriate security association. In "classical" DTLS, selecting a security association of an incoming DTLS record is accomplished with the help of the 5-tuple. If the source IP address and/or source port changes during the lifetime of an ongoing DTLS session then the receiver will be unable to locate the correct security context.

Table of Contents

1. Introduction

The Datagram Transport Layer Security (DTLS) [RFC6347] protocol was designed for securing connection-less transports, like UDP. DTLS, like TLS, starts with a handshake, which can be computationally demanding (particularly when public key cryptography is used). After a successful handshake, symmetric key cryptography is used to apply data origin authentication, integrity and confidentiality protection. This two-step approach allows endpoints to amortize the cost of the initial handshake across subsequent application data protection. Ideally, the second phase where application data is protected lasts over a long period of time since the established keys will only need to be updated once the key lifetime expires.

In DTLS as specified in RFC 6347, the IP address and port of the peer are used to identify the DTLS association. Unfortunately, in some cases, such as NAT rebinding, these values are insufficient. This is a particular issue in the Internet of Things when devices enter extended sleep periods to increase their battery lifetime. The NAT rebinding leads to connection failure, with the resulting cost of a new handshake.

This document defines an extension to DTLS 1.2 to add a CID to the DTLS record layer. The presence of the CID is negotiated via a DTLS extension.

2. Conventions and Terminology

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 RFC 2119 [RFC2119].

This document assumes familiarity with DTLS 1.2 [RFC6347].

3. The "connection_id" Extension

This document defines the "connection_id" extension, which is used in ClientHello and ServerHello messages.

The extension type is specified as follows.

```
  enum {
     connection_id(TBD1), (65535)
  } ExtensionType;
```

The extension_data field of this extension, when included in the ClientHello, MUST contain the ConnectionId structure. This structure contains the CID value the client wishes the server to use when sending messages to the client. A zero-length CID value indicates that the client is prepared to send with a CID but does not wish the server to use one when sending. Alternatively, this can be interpreted as the client wishes the server to use a zero-length CID; the result is the same.

```
struct {
    opaque cid<0..2^8-1>;
} ConnectionId;
```

A server willing to use CIDs will respond with a "connection_id" extension in the ServerHello, containing the CID it wishes the client to use when sending messages towards it. A zero-length value indicates that the server will send with the client's CID but does not wish the client to include a CID (or again, alternately, to use a zero-length CID).

Because each party sends the value in the "connection_id" extension it wants to receive as a CID in encrypted records, it is possible for an endpoint to use a globally constant length for such connection identifiers. This can in turn ease parsing and connection lookup, for example by having the length in question be a compile-time constant. Such implementations MUST still be able to send CIDs of different length to other parties. Implementations that want to use variable-length CIDs are responsible for constructing the CID in such a way that its length can be determined on reception. Note that there is no CID length information included in the record itself.

In DTLS 1.2, CIDs are exchanged at the beginning of the DTLS session only. There is no dedicated "CID update" message that allows new CIDs to be established mid-session, because DTLS 1.2 in general does not allow TLS 1.3-style post-handshake messages that do not themselves begin other handshakes. When a DTLS session is resumed or renegotiated, the "connection_id" extension is negotiated afresh.

If DTLS peers have not negotiated the use of CIDs then the RFC 6347-defined record format and content type MUST be used.

If DTLS peers have negotiated the use of a CIDs using the ClientHello and the ServerHello messages then the peers need to take the following steps.

The DTLS peers determine whether incoming and outgoing messages need to use the new record format, i.e., the record format containing the CID. The new record format with the the tls12_cid content type is only used once encryption is enabled. Plaintext payloads never use the new record type and the CID content type.

For sending, if a zero-length CID has been negotiated then the RFC 6347-defined record format and content type MUST be used (see Section 4.1 of [RFC6347]) else the new record layer format with the tls12_cid content type defined in Figure 3 MUST be used.

When transmitting a datagram with the tls12_cid content type, the new MAC computation defined in Section 5 MUST be used.

For receiving, if the tls12_cid content type is set, then the CID is used to look up the connection and the security association. If the tls12_cid content type is not set, then the connection and security association is looked up by the 5-tuple and a check MUST be made to determine whether the expected CID value is indeed zero length. If the check fails, then the datagram MUST be dropped.

When receiving a datagram with the tls12_cid content type, the new MAC computation defined in Section 5 MUST be used. When receiving a datagram with the RFC 6347-defined record format the MAC calculation defined in Section 4.1.2 of [RFC6347] MUST be used.

4. Record Layer Extensions

This specification defines the DTLS 1.2 record layer format and [I-D.ietf-tls-dtls13] specifies how to carry the CID in DTLS 1.3.

To allow a receiver to determine whether a record has a CID or not, connections which have negotiated this extension use a distinguished record type tls12_cid(TBD2). Use of this content type has the following three implications:

- The CID field is present and contains one or more bytes.

- The MAC calculation follows the process described in Section 5.

- The true content type is inside the encryption envelope, as described below.

Plaintext records are not impacted by this extension. Hence, the format of the DTLSPlaintext structure is left unchanged, as shown in Figure 1.

```
struct {
    ContentType type;
    ProtocolVersion version;
    uint16 epoch;
    uint48 sequence_number;
    uint16 length;
    opaque fragment[DTLSPlaintext.length];
} DTLSPlaintext;
```

Figure 1: DTLS 1.2 Plaintext Record Payload.

When CIDs are being used, the content to be sent is first wrapped along with its content type and optional padding into a DTLSInnerPlaintext structure. This newly introduced structure is shown in Figure 2. The DTLSInnerPlaintext byte sequence is then encrypted. To create the DTLSCiphertext structure shown in Figure 3 the CID is added.

```
struct {
    opaque content[length];
    ContentType real_type;
    uint8 zeros[length_of_padding];
} DTLSInnerPlaintext;
```

Figure 2: New DTLSInnerPlaintext Payload Structure.

content Corresponds to the fragment of a given length.

real_type The content type describing the payload.

zeros An arbitrary-length run of zero-valued bytes may appear in the cleartext after the type field. This provides an opportunity for senders to pad any DTLS record by a chosen amount as long as the total stays within record size limits. See Section 5.4 of [RFC8446] for more details. (Note that the term TLSInnerPlaintext in RFC 8446 refers to DTLSInnerPlaintext in this specification.)

```
struct {
    ContentType outer_type = tls12_cid;
    ProtocolVersion version;
    uint16 epoch;
    uint48 sequence_number;
    opaque cid[cid_length];               // New field
    uint16 length;
    opaque enc_content[DTLSCiphertext.length];
} DTLSCiphertext;
```

Figure 3: DTLS 1.2 CID-enhanced Ciphertext Record.

outer_type The outer content type of a DTLSCiphertext record carrying a CID is always set to tls12_cid(TBD2). The real content type of the record is found in DTLSInnerPlaintext.real_type after decryption.

cid The CID value, cid_length bytes long, as agreed at the time the extension has been negotiated. Recall that (as discussed previously) each peer chooses the CID value it will receive and use to identify the connection, so an implementation can choose to always recieve CIDs of a fixed length. If, however, an implementation chooses to receive different lengths of CID, the assigned CID values must be self-delineating since there is no other mechanism available to determine what connection (and thus, what CID length) is in use.

enc_content The encrypted form of the serialized DTLSInnerPlaintext structure.

All other fields are as defined in RFC 6347.

5. Record Payload Protection

Several types of ciphers have been defined for use with TLS and DTLS and the MAC calculations for those ciphers differ slightly.

This specification modifies the MAC calculation as defined in [RFC6347] and [RFC7366], as well as the definition of the additional data used with AEAD ciphers provided in [RFC6347], for records with content type tls12_cid. The modified algorithm MUST NOT be applied to records that do not carry a CID, i.e., records with content type other than tls12_cid.

The following fields are defined in this document; all other fields are as defined in the cited documents.

cid Value of the negotiated CID (variable length).

cid_length 1 byte field indicating the length of the negotiated CID.

length_of_DTLSInnerPlaintext The length (in bytes) of the serialised DTLSInnerPlaintext (two-byte integer).
The length MUST NOT exceed 2^14.

Note "+" denotes concatenation.

5.1. Block Ciphers

The following MAC algorithm applies to block ciphers that do not use the with Encrypt-then-MAC processing described in [RFC7366].

```
MAC(MAC_write_key, seq_num +
    tls12_cid +
    DTLSCiphertext.version +
    cid +
    cid_length +
    length_of_DTLSInnerPlaintext +
    DTLSInnerPlaintext.content +
    DTLSInnerPlaintext.real_type +
    DTLSInnerPlaintext.zeros
)
```

5.2. Block Ciphers with Encrypt-then-MAC processing

The following MAC algorithm applies to block ciphers that use the with Encrypt-then-MAC processing described in [RFC7366].

```
MAC(MAC_write_key, seq_num +
    tls12_cid +
    DTLSCipherText.version +
    cid +
    cid_length +
    length of (IV + DTLSCiphertext.enc_content) +
    IV +
    DTLSCiphertext.enc_content);
```

5.3. AEAD Ciphers

For ciphers utilizing authenticated encryption with additional data the following modification is made to the additional data calculation.

```
additional_data = seq_num +
                  tls12_cid +
                  DTLSCipherText.version +
                  cid +
                  cid_length +
                  length_of_DTLSInnerPlaintext;
```

6. Peer Address Update

When a record with a CID is received that has a source address different than the one currently associated with the DTLS connection, the receiver MUST NOT replace the address it uses for sending records to its peer with the source address specified in the received datagram unless the following three conditions are met:

- The received datagram has been cryptographically verified using the DTLS record layer processing procedures.

- The received datagram is "newer" (in terms of both epoch and sequence number) than the newest datagram received. Reordered datagrams that are sent prior to a change in a peer address might otherwise cause a valid address change to be reverted. This also limits the ability of an attacker to use replayed datagrams to force a spurious address change, which could result in denial of service. An attacker might be able to succeed in changing a peer address if they are able to rewrite source addresses and if replayed packets are able to arrive before any original.

- There is a strategy for ensuring that the new peer address is able to receive and process DTLS records. No such test is defined in this specification.

The conditions above are necessary to protect against attacks that use datagrams with spoofed addresses or replayed datagrams to trigger attacks. Note that there is no requirement for use of the anti-replay window mechanism defined in Section 4.1.2.6 of DTLS 1.2. Both solutions, the "anti-replay window" or "newer" algorithm, will prevent address updates from replay attacks while the latter will only apply to peer address updates and the former applies to any application layer traffic.

Note that datagrams that pass the DTLS cryptographic verification procedures but do not trigger a change of peer address are still valid DTLS records and are still to be passed to the application.

Application protocols that implement protection against these attacks depend on being aware of changes in peer addresses so that they can engage the necessary mechanisms. When delivered such an event, an application layer-specific address validation mechanism can be triggered, for example one that is based on successful exchange of a minimal amount of ping-pong traffic with the peer. Alternatively, an DTLS-specific mechanism may be used, as described in [I-D.tschofenig-tls-dtls-rrc].

DTLS implementations MUST silently discard records with bad MACs or that are otherwise invalid.

7. Examples

Figure 4 shows an example exchange where a CID is used uni-directionally from the client to the server. To indicate that a zero-length CID is present in the "connection_id" extension we use the notation 'connection_id=empty'.

```
   Client                                             Server
   ------                                             ------

   ClientHello                 -------->
   (connection_id=empty)


                               <--------      HelloVerifyRequest
                                                        (cookie)

   ClientHello                 -------->
   (connection_id=empty)
   (cookie)

                                                     ServerHello
                                             (connection_id=100)
                                                     Certificate
                                               ServerKeyExchange
                                              CertificateRequest
                               <--------         ServerHelloDone

   Certificate
   ClientKeyExchange
   CertificateVerify
   [ChangeCipherSpec]
   Finished                    -------->
   <CID=100>

                                              [ChangeCipherSpec]
                               <--------                Finished


   Application Data            ========>
   <CID=100>

                               <========        Application Data

   Legend:

   <...> indicates that a connection id is used in the record layer
   (...) indicates an extension
   [...] indicates a payload other than a handshake message
```

Figure 4: Example DTLS 1.2 Exchange with CID

Note: In the example exchange the CID is included in the record layer once encryption is enabled. In DTLS 1.2 only one handshake message is encrypted, namely the Finished message. Since the example shows how to use the CID for payloads sent from the client to the server, only the record layer payloads containing the Finished message or application data include a CID.

8. Privacy Considerations

The CID replaces the previously used 5-tuple and, as such, introduces an identifier that remains persistent during the lifetime of a DTLS connection. Every identifier introduces the risk of linkability, as explained in [RFC6973].

An on-path adversary observing the DTLS protocol exchanges between the DTLS client and the DTLS server is able to link the observed payloads to all subsequent payloads carrying the same ID pair (for bi-directional communication). Without multi-homing or mobility, the use of the CID exposes the same information as the 5-tuple.

With multi-homing, a passive attacker is able to correlate the communication interaction over the two paths and the sequence number makes it possible to correlate packets across CID changes. The lack of a CID update mechanism in DTLS 1.2 makes this extension unsuitable for mobility scenarios where correlation must be considered. Deployments that use DTLS in multi-homing environments and are concerned about this aspects SHOULD refuse to use CIDs in DTLS 1.2 and switch to DTLS 1.3 where a CID update mechanism is provided and sequence number encryption is available.

The specification introduces record padding for the CID-enhanced record layer, which is a privacy feature not available with the original DTLS 1.2 specification. Padding allows to inflate the size of the ciphertext making traffic analysis more difficult. More details about record padding can be found in Section 5.4 and Appendix E.3 of RFC 8446.

Finally, endpoints can use the CID to attach arbitrary per-connection metadata to each record they receive on a given connection. This may be used as a mechanism to communicate per-connection information to on-path observers. There is no straightforward way to address this concern with CIDs that contain arbitrary values. Implementations concerned about this aspect SHOULD refuse to use CIDs.

9. Security Considerations

An on-path adversary can create reflection attacks against third parties because a DTLS peer has no means to distinguish a genuine address update event (for example, due to a NAT rebinding) from one that is malicious. This attack is of concern when there is a large asymmetry of request/response message sizes.

Additionally, an attacker able to observe the data traffic exchanged between two DTLS peers is able to replay datagrams with modified IP address/port numbers.

The topic of peer address updates is discussed in Section 6.

10. IANA Considerations

IANA is requested to allocate an entry to the existing TLS "ExtensionType Values" registry, defined in [RFC5246], for connection_id(TBD1) as described in the table below. IANA is requested to add an extra column to the TLS ExtensionType Values registry to indicate whether an extension is only applicable to DTLS and to include this document as an additional reference for the registry.

Value	Extension Name	TLS 1.3	DTLS Only	Recommended	Reference
TBD1	connection_id	CH, SH	Y	N	[[This doc]]

Note: The value "N" in the Recommended column is set because this extension is intended only for specific use cases. This document describes the behavior of this extension for DTLS 1.2 only; it is not applicable to TLS, and its usage for DTLS 1.3 is described in [I-D.ietf-tls-dtls13].

IANA is requested to allocate tls12_cid(TBD2) in the "TLS ContentType Registry". The tls12_cid ContentType is only applicable to DTLS 1.2.

11. References

11.1. Normative References

[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>.

[RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, DOI 10.17487/RFC5246, August 2008, <https://www.rfc-editor.org/info/rfc5246>.

[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/info/rfc6347>.

[RFC7366] Gutmann, P., "Encrypt-then-MAC for Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS)", RFC 7366, DOI 10.17487/RFC7366, September 2014, <https://www.rfc-editor.org/info/rfc7366>.

[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>.

11.2. Informative References

[I-D.ietf-tls-dtls13] Rescorla, E., Tschofenig, H., and N. Modadugu, "The Datagram Transport Layer Security (DTLS) Protocol Version 1.3", draft-ietf-tls-dtls13-38 (work in progress), May 2020.

[I-D.tschofenig-tls-dtls-rrc] Fossati, T. and H. Tschofenig, "Return Routability Check for DTLS 1.2 and DTLS 1.3", draft-tschofenig-tls-dtls-rrc-01 (work in progress), March 2020.

[RFC6973] Cooper, A., Tschofenig, H., Aboba, B., Peterson, J., Morris, J., Hansen, M., and R. Smith, "Privacy Considerations for Internet Protocols", RFC 6973, DOI 10.17487/RFC6973, July 2013, <https://www.rfc-editor.org/info/rfc6973>.

11.3. URIs

[1] mailto:tls@ietf.org

[2] https://www1.ietf.org/mailman/listinfo/tls

[3] https://www.ietf.org/mail-archive/web/tls/current/index.html

Appendix A. History

RFC EDITOR: PLEASE REMOVE THE THIS SECTION

draft-ietf-tls-dtls-connection-id-08

- RRC draft moved from normative to informative.

draft-ietf-tls-dtls-connection-id-07

- Wording changes in the security and privacy consideration and the peer address update sections.

draft-ietf-tls-dtls-connection-id-06

- Updated IANA considerations

- Enhanced security consideration section to describe a potential man-in-the-middle attack concerning address validation.

draft-ietf-tls-dtls-connection-id-05

- Restructed Section 5 "Record Payload Protection"

draft-ietf-tls-dtls-connection-id-04

- Editorial simplifications to the 'Record Layer Extensions' and the 'Record Payload Protection' sections.

- Added MAC calculations for block ciphers with and without Encrypt-then-MAC processing.

draft-ietf-tls-dtls-connection-id-03

- Updated list of contributors

- Updated list of contributors and acknowledgements

- Updated example

- Changed record layer design

- Changed record payload protection

- Updated introduction and security consideration section

- Author- and affiliation changes

draft-ietf-tls-dtls-connection-id-02

- Move to internal content types a la DTLS 1.3.

draft-ietf-tls-dtls-connection-id-01

- Remove 1.3 based on the WG consensus at IETF 101

draft-ietf-tls-dtls-connection-id-00

- Initial working group version (containing a solution for DTLS 1.2 and 1.3)

draft-rescorla-tls-dtls-connection-id-00

- Initial version

Appendix B. Working Group Information

RFC EDITOR: PLEASE REMOVE THE THIS SECTION

The discussion list for the IETF TLS working group is located at the e-mail address tls@ietf.org [1]. Information on the group and information on how to subscribe to the list is at https://www1.ietf.org/mailman/listinfo/tls [2]

Archives of the list can be found at: https://www.ietf.org/mail-archive/web/tls/current/index.html [3]

Appendix C. Contributors

Many people have contributed to this specification and we would like to thank the following individuals for their contributions:

* Yin Xinxing
  Huawei
  yinxinxing@huawei.com

* Nikos Mavrogiannopoulos
  RedHat
  nmav@redhat.com

* Tobias Gondrom
  tobias.gondrom@gondrom.org

Additionally, we would like to thank the Connection ID task force team members:

- Martin Thomson (Mozilla)

- Christian Huitema (Private Octopus Inc.)

- Jana Iyengar (Google)

- Daniel Kahn Gillmor (ACLU)

- Patrick McManus (Mozilla)

- Ian Swett (Google)

- Mark Nottingham (Fastly)

The task force team discussed various design ideas, including cryptographically generated session ids using hash chains and public key encryption, but dismissed them due to their inefficiency. The approach described in this specification is the simplest possible design that works given the limitations of DTLS 1.2. DTLS 1.3 provides better privacy features and developers are encouraged to switch to the new version of DTLS.

Finally, we want to thank the IETF TLS working group chairs, Chris Wood, Joseph Salowey, and Sean Turner, for their patience, support and feedback.

Appendix D. Acknowledgements

We would like to thank Achim Kraus for his review comments and implementation feedback.

Authors' Addresses

Eric Rescorla (editor)
RTFM, Inc.

EMail: ekr@rtfm.com

Hannes Tschofenig (editor)
Arm Limited

EMail: hannes.tschofenig@arm.com

Thomas Fossati
Arm Limited

EMail: thomas.fossati@arm.com

tls
Internet-Draft
Intended status: Standards Track
Expires: 19 April 2021

E. Rescorla
RTFM, Inc.
K. Oku
Fastly
N. Sullivan
C.A. Wood
Cloudflare

16 October 2020

TLS Encrypted Client Hello
draft-ietf-tls-esni-08

Abstract

This document describes a mechanism in Transport Layer Security (TLS) for encrypting a ClientHello message under a server public key.

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

DISCLAIMER: This is very early a work-in-progress design and has not yet seen significant (or really any) security analysis. It should not be used as a basis for building production systems.

Although TLS 1.3 [RFC8446] encrypts most of the handshake, including the server certificate, there are several ways in which an on-path attacker can learn private information about the connection. The plaintext Server Name Indication (SNI) extension in ClientHello messages, which leaks the target domain for a given connection, is perhaps the most sensitive information unencrypted in TLS 1.3.

The target domain may also be visible through other channels, such as plaintext client DNS queries, visible server IP addresses (assuming the server does not use domain-based virtual hosting), or other indirect mechanisms such as traffic analysis. DoH [RFC8484] and DPRIVE [RFC7858] [RFC8094] provide mechanisms for clients to conceal DNS lookups from network inspection, and many TLS servers host multiple domains on the same IP address. In such environments, the SNI remains the primary explicit signal used to determine the server's identity.

The TLS Working Group has studied the problem of protecting the SNI, but has been unable to develop a completely generic solution. [RFC8744] provides a description of the problem space and some of the proposed techniques. One of the more difficult problems is "Do not stick out" ([RFC8744], Section 3.4): if only sensitive or private services use SNI encryption, then SNI encryption is a signal that a client is going to such a service. For this reason, much recent work has focused on concealing the fact that the SNI is being protected. Unfortunately, the result often has undesirable performance consequences, incomplete coverage, or both.

The protocol specified by this document takes a different approach. It assumes that private origins will co-locate with or hide behind a provider (reverse proxy, application server, etc.) that protects sensitive ClientHello parameters, including the SNI, for all of the domains it hosts. These co-located servers form an anonymity set wherein all elements have a consistent configuration, e.g., the set of supported application protocols, ciphersuites, TLS versions, and so on. Usage of this mechanism reveals that a client is connecting to a particular service provider, but does not reveal which server from the anonymity set terminates the connection. Thus, it leaks no more than what is already visible from the server IP address.

This document specifies a new TLS extension, called Encrypted Client Hello (ECH), that allows clients to encrypt their ClientHello to a supporting server. This protects the SNI and other potentially sensitive fields, such as the ALPN list [RFC7301]. This extension is only supported with (D)TLS 1.3 [RFC8446] and newer versions of the protocol.

2. 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. All TLS notation comes from [RFC8446], Section 3.

3. Overview

This protocol is designed to operate in one of two topologies illustrated below, which we call "Shared Mode" and "Split Mode".

3.1. Topologies

```
                   +----------------------+
                   |                      |
                   |   2001:DB8::1111     |
                   |                      |
Client <----->     | private.example.org  |
                   |                      |
                   | public.example.com   |
                   |                      |
                   +----------------------+
                           Server
```

Figure 1: Shared Mode Topology

In Shared Mode, the provider is the origin server for all the domains whose DNS records point to it. In this mode, the TLS connection is terminated by the provider.

```
              +---------------------+     +----------------------+
              |                     |     |                      |
              |   2001:DB8::1111    |     |   2001:DB8::EEEE     |
Client <----------------------------->|                      |
              | public.example.com  |     | private.example.com  |
              |                     |     |                      |
              +---------------------+     +----------------------+
               Client-Facing Server            Backend Server
```

Figure 2: Split Mode Topology

In Split Mode, the provider is not the origin server for private domains. Rather, the DNS records for private domains point to the provider, and the provider’s server relays the connection back to the origin server, who terminates the TLS connection with the client. Importantly, service provider does not have access to the plaintext of the connection.

In the remainder of this document, we will refer to the ECH-service provider as the "client-facing server" and to the TLS terminator as the "backend server". These are the same entity in Shared Mode, but in Split Mode, the client-facing and backend servers are physically separated.

3.2. Encrypted ClientHello (ECH)

ECH allows the client to encrypt sensitive ClientHello extensions, e.g., SNI, ALPN, etc., under the public key of the client-facing server. This requires the client-facing server to publish the public key and metadata it uses for ECH for all the domains for which it serves directly or indirectly (via Split Mode). This document defines the format of the ECH encryption public key and metadata, referred to as an ECH configuration, and delegates DNS publication details to [HTTPS-RR], though other delivery mechanisms are possible. In particular, if some of the clients of a private server are applications rather than Web browsers, those applications might have the public key and metadata preconfigured.

When a client wants to establish a TLS session with the backend server, it constructs its ClientHello as usual (we will refer to this as the ClientHelloInner message) and then encrypts this message using the public key of the ECH configuration. It then constructs a new ClientHello (ClientHelloOuter) with innocuous values for sensitive extensions, e.g., SNI, ALPN, etc., and with an "encrypted_client_hello" extension, which this document defines (Section 5). The extension's payload carries the encrypted ClientHelloInner and specifies the ECH configuration used for encryption. Finally, it sends ClientHelloOuter to the server.

Upon receiving the ClientHelloOuter, the client-facing server takes one of the following actions:

1. If it does not support ECH, it ignores the "encrypted_client_hello" extension and proceeds with the handshake as usual, per [RFC8446], Section 4.1.2.

2. If it supports ECH but cannot decrypt the extension, then it terminates the handshake using the ClientHelloOuter. This is referred to as "ECH rejection". When ECH is rejected, the server sends an acceptable ECH configuration in its EncryptedExtensions message.

3. If it supports ECH and decrypts the extension, it forwards the ClientHelloInner to the backend, who terminates the connection. This is referred to as "ECH acceptance".

Upon receiving the server's response, the client determines whether or not ECH was accepted and proceeds with the handshake accordingly. (See Section 6 for details.)

Informally, a primary goal of ECH is ensuring that connections to servers in the same anonymity set are indistinguishable from one another without affecting any existing security properties of TLS 1.3. See Section 10.1 for more details about the ECH security and privacy goals.

4. Encrypted ClientHello Configuration

ECH uses draft-05 of HPKE for public key encryption [I-D.irtf-cfrg-hpke]. The ECH configuration is defined by the following "ECHConfigs" structure.

```
opaque HpkePublicKey<1..2^16-1>;
uint16 HpkeKemId;  // Defined in I-D.irtf-cfrg-hpke
uint16 HpkeKdfId;  // Defined in I-D.irtf-cfrg-hpke
uint16 HpkeAeadId; // Defined in I-D.irtf-cfrg-hpke

struct {
    HpkeKdfId kdf_id;
    HpkeAeadId aead_id;
} ECHCipherSuite;

struct {
    opaque public_name<1..2^16-1>;

    HpkePublicKey public_key;
    HpkeKemId kem_id;
    ECHCipherSuite cipher_suites<4..2^16-4>;

    uint16 maximum_name_length;
    Extension extensions<0..2^16-1>;
} ECHConfigContents;

struct {
    uint16 version;
    uint16 length;
    select (ECHConfig.version) {
      case 0xfe08: ECHConfigContents contents;
    }
} ECHConfig;

ECHConfig ECHConfigs<1..2^16-1>;
```

The "ECHConfigs" structure contains one or more "ECHConfig" structures in decreasing order of preference. This allows a server to support multiple versions of ECH and multiple sets of ECH parameters.

The "ECHConfig" structure contains the following fields:

version The version of ECH for which this configuration is used. Beginning with draft-08, the version is the same as the code point for the "encrypted_client_hello" extension. Clients MUST ignore any "ECHConfig" structure with a version they do not support.

length The length, in bytes, of the next field.

contents An opaque byte string whose contents depend on the version. For this specification, the contents are an "ECHConfigContents" structure.

The "ECHConfigContents" structure contains the following fields:

public_name The non-empty name of client-facing server, i.e., the entity trusted to update these encryption keys. This is used to repair misconfigurations, as described in Section 6.3.

public_key The HPKE public key used by the client to encrypt ClientHelloInner.

kem_id The HPKE KEM identifier corresponding to "public_key". Clients MUST ignore any "ECHConfig" structure with a key using a KEM they do not support.

cipher_suites The list of HPKE AEAD and KDF identifier pairs clients can use for encrypting ClientHelloInner.

maximum_name_length The largest name the server expects to support, if known. If this value is not known it can be set to zero, in which case clients SHOULD use the inner ClientHello padding scheme described below. That could happen if wildcard names are in use, or if names can be added or removed from the anonymity set during the lifetime of a particular resource record value.

extensions A list of extensions that the client must take into consideration when generating a ClientHello message. These are described below (Section 4.1).

4.1. Configuration Extensions

ECH configuration extensions are used to to provide room for additional functionality as needed. See Section 12 for guidance on which types of extensions are appropriate for this structure.

The format is as defined in [RFC8446], Section 4.2. The same interpretation rules apply: extensions MAY appear in any order, but there MUST NOT be more than one extension of the same type in the extensions block. An extension can be tagged as mandatory by using an extension type codepoint with the high order bit set to 1. A client that receives a mandatory extension they do not understand MUST reject the "ECHConfig" content.

Clients MUST parse the extension list and check for unsupported mandatory extensions. If an unsupported mandatory extension is present, clients MUST ignore the "ECHConfig".

5. The "encrypted_client_hello" Extension

The encrypted ClientHelloInner is carried in an "encrypted_client_hello" extension, defined as follows:

```
enum {
   encrypted_client_hello(0xfe08), (65535)
} ExtensionType;
```

The extension request is carried by the ClientHelloOuter, i.e., the ClientHello transmitted to the client-facing server. The payload contains the following "ClientECH" structure:

```
struct {
   ECHCipherSuite cipher_suite;
   opaque config_id<0..255>;
   opaque enc<1..2^16-1>;
   opaque payload<1..2^16-1>;
} ClientECH;
```

cipher_suite The cipher suite used to encrypt ClientHelloInner. This MUST match a value provided in the corresponding "ECHConfig.cipher_suites" list.

config_id The configuration identifier, equal to "Expand(Extract("", config), "tls ech config id", Nh)", where "config" is the "ECHConfig" structure and "Extract", "Expand", and "Nh" are as specified by the cipher suite KDF. (Passing the literal """" as the salt is interpreted by "Extract" as no salt being provided.) The length of this value SHOULD NOT be less than 16 bytes unless it is optional for an application; see Section 10.4.

enc The HPKE encapsulated key, used by servers to decrypt the corresponding "payload" field.

payload The serialized and encrypted ClientHelloInner structure, encrypted using HPKE as described in Section 6.1.

When offering the "encrypted_client_hello" extension in its ClientHelloOuter, the client MUST also offer an empty "encrypted_client_hello" extension in its ClientHelloInner, wherever applicable. (This requirement is not applicable when the extension is generated as described in Section 6.4.)

When the client offers the "encrypted_client_hello" extension, the server MAY include an "encrypted_client_hello" extension in its EncryptedExtensions message with the following payload:

```
struct {
   ECHConfigs retry_configs;
} ServerECH;
```

retry_configs An ECHConfigs structure containing one or more ECHConfig structures, in decreasing order of preference, to be used by the client in subsequent connection attempts.

This document also defines the "ech_required" alert, which clients MUST send when it offered an "encrypted_client_hello" extension that was not accepted by the server. (See Section 11.2.)

5.1. Encoding the ClientHelloInner

Some TLS 1.3 extensions can be quite large and having them both in ClientHelloInner and ClientHelloOuter will lead to a very large overall size. One particularly pathological example is "key_share" with post-quantum algorithms. In order to reduce the impact of duplicated extensions, the client may use the "outer_extensions" extension.

```
enum {
   outer_extensions(0xfd00), (65535)
} ExtensionType;

ExtensionType OuterExtensions<2..254>;
```

OuterExtensions consists of one or more ExtensionType values, each of which reference an extension in ClientHelloOuter.

When sending ClientHello, the client first computes ClientHelloInner, including any PSK binders. It then computes a new value, the EncodedClientHelloInner, by first making a copy of ClientHelloInner. It then replaces the legacy_session_id field with an empty string.

The client then MAY substitute extensions which it knows will be duplicated in ClientHelloOuter. To do so, the client removes and replaces extensions from EncodedClientHelloInner with a single "outer_extensions" extension. Removed extensions MUST be ordered consecutively in ClientHelloInner. The list of outer extensions, OuterExtensions, includes those which were removed from EncodedClientHelloInner, in the order in which they were removed.

Finally, EncodedClientHelloInner is serialized as a ClientHello structure, defined in Section 4.1.2 of [RFC8446]. Note this does not include the four-byte header included in the Handshake structure.

The client-facing server computes ClientHelloInner by reversing this process. First it makes a copy of EncodedClientHelloInner and copies the legacy_session_id field from ClientHelloOuter. It then looks for an "outer_extensions" extension. If found, it replaces the extension with the corresponding sequence of extensions in the ClientHelloOuter. If any referenced extensions are missing or if "encrypted_client_hello" appears in the list, the server MUST abort the connection with an "illegal_parameter" alert.

The "outer_extensions" extension is only used for compressing the ClientHelloInner. It MUST NOT be sent in either ClientHelloOuter or ClientHelloInner.

5.2. Authenticating the ClientHelloOuter

To prevent a network attacker from modifying the reconstructed ClientHelloInner (see Section 10.10.3), ECH authenticates ClientHelloOuter by deriving a ClientHelloOuterAAD value. This is computed by serializing ClientHelloOuter with the "encrypted_client_hello" extension removed. ClientHelloOuterAAD is then passed as the associated data parameter to the HPKE encryption.

Note the decompression process in Section 5.1 forbids "encrypted_client_hello" in OuterExtensions. This ensures the unauthenticated portion of ClientHelloOuter is not incorporated into ClientHelloInner.

6. Client Behavior

6.1. Sending an Encrypted ClientHello

To offer ECH, the client first chooses a suitable ECH configuration. To determine if a given "ECHConfig" is suitable, it checks that it supports the KEM algorithm identified by "ECHConfig.kem_id" and at least one KDF/AEAD algorithm identified by "ECHConfig.cipher_suites". Once a suitable configuration is found, the client selects the cipher suite it will use for encryption. It MUST NOT choose a cipher suite not advertised by the configuration.

Next, the client constructs the ClientHelloInner message just as it does a standard ClientHello, with the exception of the following rules:

1. It MUST NOT offer to negotiate TLS 1.2 or below. Note this is necessary to ensure the backend server does not negotiate a TLS version that is incompatible with ECH.

2. It MUST NOT offer to resume any session for TLS 1.2 and below.

3. It SHOULD contain TLS padding [RFC7685] as described in Section 6.2.

4. If it intends to compress any extensions (see Section 5.1), it MUST order those extensions consecutively.

The client then constructs EncodedClientHelloInner as described in Section 5.1. Finally, it constructs the ClientHelloOuter message just as it does a standard ClientHello, with the exception of the following rules:

1. It MUST offer to negotiate TLS 1.3 or above.

2. If it compressed any extensions in EncodedClientHelloInner, it MUST copy the corresponding extensions from ClientHelloInner.

3. It MAY copy any other field from the ClientHelloInner except ClientHelloInner.random. Instead, It MUST generate a fresh ClientHelloOuter.random using a secure random number generator. (See Section 10.10.1.)

4. It MUST copy the legacy_session_id field from ClientHelloInner. This allows the server to echo the correct session ID for TLS 1.3’s compatibility mode (see Appendix D.4 of [RFC8446]) when ECH is negotiated.

5. It MUST include an "encrypted_client_hello" extension with a payload constructed as described below.

6. The value of "ECHConfig.public_name" MUST be placed in the "server_name" extension.

7. It MUST NOT include the "pre_shared_key" extension. (See Section 10.10.3.)

The client might duplicate non-sensitive extensions in both messages. However, implementations need to take care to ensure that sensitive extensions are not offered in the ClientHelloOuter. See Section 10.5 for additional guidance.

To encrypt EncodedClientHelloInner, the client first computes ClientHelloOuterAAD as described in Section 5.2. Note this requires the "encrypted_client_hello" be computed after all other extensions. In particular, this is possible because the "pre_shared_key" extension is forbidden in ClientHelloOuter.

The client then generates the HPKE encryption context. Finally, it computes the encapsulated key, context, HRR key (see Section 6.3.3), and payload as:

```
pkR = Deserialize(ECHConfig.public_key)
enc, context = SetupBaseS(pkR,
                          "tls ech" || 0x00 || ECHConfig)
ech_hrr_key = context.Export("tls ech hrr key", 32)
payload = context.Seal(ClientHelloOuterAAD,
                       EncodedClientHelloInner)
```

Note that the HPKE functions Deserialize and SetupBaseS are those which match "ECHConfig.kem_id" and the AEAD/KDF used with "context" are those which match the client's chosen preference from "ECHConfig.cipher_suites". The "info" parameter to SetupBaseS is the concatenation of "tls ech", a zero byte, and the serialized ECHConfig.

The value of the "encrypted_client_hello" extension in the ClientHelloOuter is a "ClientECH" with the following values:

* "cipher_suite", the client's chosen cipher suite;

* "config_id", the identifier of the chosen ECHConfig structure;

* "enc", as computed above; and

* "payload", as computed above.

If optional configuration identifiers (see Section 10.4)) are used, the "config_id" field MAY be empty or randomly generated. Unless specified by the application using (D)TLS or externally configured on both sides, implementations MUST compute the field as specified in Section 5.

6.2. Recommended Padding Scheme

This section describes a deterministic padding mechanism based on the following observation: individual extensions can reveal sensitive information through their length. Thus, each extension in the inner ClientHello may require different amounts of padding. This padding may be fully determined by the client's configuration or may require server input.

By way of example, clients typically support a small number of application profiles. For instance, a browser might support HTTP with ALPN values ["http/1.1, "h2"] and WebRTC media with ALPNs ["webrtc", "c-webrtc"]. Clients SHOULD pad this extension by rounding up to the total size of the longest ALPN extension across all application profiles. The target padding length of most ClientHello extensions can be computed in this way.

In contrast, clients do not know the longest SNI value in the client-facing server's anonymity set without server input. For the "server_name" extension with length D, clients SHOULD use the server's length hint L (ECHCOnfig.maximum_name_length) when computing the padding as follows:

1. If L >= D, add L - D bytes of padding. This rounds to the server's advertised hint, i.e., ECHConfig.maximum_name_length.

2. Otherwise, let P = 31 - ((D - 1) % 32), and add P bytes of padding, plus an additional 32 bytes if D + P < L + 32. This rounds D up to the nearest multiple of 32 bytes that permits at least 32 bytes of length ambiguity.

In addition to padding ClientHelloInner, clients and servers will also need to pad all other handshake messages that have sensitive-length fields. For example, if a client proposes ALPN values in ClientHelloInner, the server-selected value will be returned in an EncryptedExtension, so that handshake message also needs to be padded using TLS record layer padding.

6.3. Handling the Server Response

As described in Section 7, the server MAY either accept ECH and use ClientHelloInner or reject it and use ClientHelloOuter. In handling the server's response, the client's first step is to determine which value was used. The client presumes acceptance if the last 8 bytes of ServerHello.random are equal to "accept_confirmation" as defined in Section 7.2. Otherwise, it presumes rejection.

6.3.1. Accepted ECH

If the server used ClientHelloInner, the client proceeds with the connection as usual, authenticating the connection for the origin server.

6.3.2. Rejected ECH

If the server used ClientHelloOuter, the client proceeds with the handshake, authenticating for ECHConfig.public_name as described in Section 6.3.2.1. If authentication or the handshake fails, the client MUST return a failure to the calling application. It MUST NOT use the retry keys.

Otherwise, when the handshake completes successfully with the public name authenticated, the client MUST abort the connection with an "ech_required" alert. It then processes the "retry_configs" field from the server's "encrypted_client_hello" extension.

If one of the values contains a version supported by the client, it can regard the ECH keys as securely replaced by the server. It SHOULD retry the handshake with a new transport connection, using that value to encrypt the ClientHello. The value may only be applied to the retry connection. The client MUST continue to use the previously-advertised keys for subsequent connections. This avoids introducing pinning concerns or a tracking vector, should a malicious server present client-specific retry keys to identify clients.

If none of the values provided in "retry_configs" contains a supported version, the client can regard ECH as securely disabled by the server. As below, it SHOULD then retry the handshake with a new transport connection and ECH disabled.

If the field contains any other value, the client MUST abort the connection with an "illegal_parameter" alert.

If the server negotiates an earlier version of TLS, or if it does not provide an "encrypted_client_hello" extension in EncryptedExtensions, the client proceeds with the handshake, authenticating for

ECHConfigContents.public_name as described in Section 6.3.2.1. If an earlier version was negotiated, the client MUST NOT enable the False Start optimization [RFC7918] for this handshake. If authentication or the handshake fails, the client MUST return a failure to the calling application. It MUST NOT treat this as a secure signal to disable ECH.

Otherwise, when the handshake completes successfully with the public name authenticated, the client MUST abort the connection with an "ech_required" alert. The client can then regard ECH as securely disabled by the server. It SHOULD retry the handshake with a new transport connection and ECH disabled.

Clients SHOULD implement a limit on retries caused by "ech_retry_request" or servers which do not acknowledge the "encrypted_client_hello" extension. If the client does not retry in either scenario, it MUST report an error to the calling application.

6.3.2.1. Authenticating for the Public Name

When the server rejects ECH or otherwise ignores "encrypted_client_hello" extension, it continues with the handshake using the plaintext "server_name" extension instead (see Section 7). Clients that offer ECH then authenticate the connection with the public name, as follows:

* The client MUST verify that the certificate is valid for ECHConfigContents.public_name. If invalid, it MUST abort the connection with the appropriate alert.

* If the server requests a client certificate, the client MUST respond with an empty Certificate message, denoting no client certificate.

Note that authenticating a connection for the public name does not authenticate it for the origin. The TLS implementation MUST NOT report such connections as successful to the application. It additionally MUST ignore all session tickets and session IDs presented by the server. These connections are only used to trigger retries, as described in Section 6.3. This may be implemented, for instance, by reporting a failed connection with a dedicated error code.

6.3.3. HelloRetryRequest

If the server sends a HelloRetryRequest in response to the ClientHello, the client sends a second updated ClientHello per the rules in [RFC8446]. However, at this point, the client does not know whether the server processed ClientHelloOuter or ClientHelloInner, and MUST regenerate both values to be acceptable. Note: if ClientHelloOuter and ClientHelloInner use different groups for their key shares or differ in some other way, then the HelloRetryRequest may actually be invalid for one or the other ClientHello, in which case a fresh ClientHello MUST be generated, ignoring the instructions in HelloRetryRequest. Otherwise, the usual rules for HelloRetryRequest processing apply.

Clients bind encryption of the second ClientHelloInner to encryption of the first ClientHelloInner via the derived ech_hrr_key by modifying HPKE setup as follows:

```
pkR = Deserialize(ECHConfig.public_key)
enc, context = SetupPSKS(pkR, "tls ech" || 0x00 || ECHConfig,
                         ech_hrr_key, "hrr key")
```

The "info" parameter to SetupPSKS is the concatenation of "tls ech", a zero byte, and the serialized ECHConfig. Clients then encrypt the second ClientHelloInner using this new HPKE context. In doing so, the encrypted value is also authenticated by ech_hrr_key. The rationale for this is described in Section 10.10.2.

Client-facing servers perform the corresponding process when decrypting second ClientHelloInner messages. In particular, upon receipt of a second ClientHello message with a ClientECH value, servers set up their HPKE context and decrypt ClientECH as follows:

```
context = SetupPSKR(ClientECH.enc, skR,
    "tls ech" || 0x00 || ECHConfig, ech_hrr_key, "hrr key")
EncodedClientHelloInner = context.Open(ClientHelloOuterAAD,
                                       ClientECH.payload)
```

ClientHelloOuterAAD is computed from the second ClientHelloOuter as described in Section 5.2. The "info" parameter to SetupPSKR is computed as above.

If the client offered ECH in the first ClientHello, then it MUST offer ECH in the second. Likewise, if the client did not offer ECH in the first ClientHello, then it MUST NOT not offer ECH in the second.

[[OPEN ISSUE: Should we be using the PSK input or the info input? On the one hand, the requirements on info seem weaker, but maybe actually this needs to be secret? Analysis needed.]]

6.4. GREASE Extensions

If the client attempts to connect to a server and does not have an ECHConfig structure available for the server, it SHOULD send a GREASE [RFC8701] "encrypted_client_hello" extension as follows:

* Set the "suite" field to a supported ECHCipherSuite. The selection SHOULD vary to exercise all supported configurations, but MAY be held constant for successive connections to the same server in the same session.

* Set the "config_id" field to a randomly-generated string of "Nh" bytes, where "Nh" is the output length of the "Extract" function of the KDF associated with the chosen cipher suite. (The KDF API is specified in [I-D.irtf-cfrg-hpke].)

* Set the "enc" field to a randomly-generated valid encapsulated public key output by the HPKE KEM.

* Set the "payload" field to a randomly-generated string of L+C bytes, where C is the ciphertext expansion of selected AEAD scheme and L is the size of the ClientHelloInner message the client would use given an ECHConfig structure, padded according to Section 6.2.

If the server sends an "encrypted_client_hello" extension, the client MUST check the extension syntactically and abort the connection with a "decode_error" alert if it is invalid. It otherwise ignores the extension and MUST NOT use the retry keys.

[[OPEN ISSUE: if the client sends a GREASE "encrypted_client_hello" extension, should it also send a GREASE "pre_shared_key" extension? If not, GREASE+ticket is a trivial distinguisher.]]

Offering a GREASE extension is not considered offering an encrypted ClientHello for purposes of requirements in Section 6. In particular, the client MAY offer to resume sessions established without ECH.

7. Server Behavior

7.1. Client-Facing Server

Upon receiving an "encrypted_client_hello" extension, the client-facing server determines if it will accept ECH, prior to negotiating any other TLS parameters. Note that successfully decrypting the extension will result in a new ClientHello to process, so even the client's TLS version preferences may have changed.

First, the server collects a set of candidate ECHConfigs. This set is determined by one of the two following methods:

1. Compare ClientECH.config_id against identifiers of known ECHConfigs and select the one that matches, if any, as a candidate.

2. Collect all known ECHConfigs as candidates, with trial decryption below determining the final selection.

Some uses of ECH, such as local discovery mode, may omit the ClientECH.config_id since it can be used as a tracking vector. In such cases, the second method should be used for matching ClientECH to known ECHConfig. See Section 10.4. Unless specified by the application using (D)TLS or externally configured on both sides, implementations MUST use the first method.

The server then iterates over all candidate ECHConfigs, attempting to decrypt the "encrypted_client_hello" extension:

The server verifies that the ECHConfig supports the cipher suite indicated by the ClientECH.cipher_suite and that the version of ECH indicated by the client matches the ECHConfig.version. If not, the server continues to the next candidate ECHConfig.

Next, the server decrypts ClientECH.payload, using the private key skR corresponding to ECHConfig, as follows:

```
context = SetupBaseR(ClientECH.enc, skR,
                     "tls ech" || 0x00 || ECHConfig)
EncodedClientHelloInner = context.Open(ClientHelloOuterAAD,
                                       ClientECH.payload)
ech_hrr_key = context.Export("tls ech hrr key", 32)
```

ClientHelloOuterAAD is computed from ClientHelloOuter as described in Section 5.2. The "info" parameter to SetupBaseS is the concatenation "tls ech", a zero byte, and the serialized ECHConfig. If decryption fails, the server continues to the next candidate ECHConfig. Otherwise, the server reconstructs ClientHelloInner from EncodedClientHelloInner, as described in Section 5.1. It then stops consider candidate ECHConfigs.

Upon determining the ClientHelloInner, the client-facing server then forwards the ClientHelloInner to the appropriate backend server, which proceeds as in Section 7.2. If the backend server responds with a HelloRetryRequest, the client-facing server forwards it, decrypts the client's second ClientHelloOuter using the modified procedure in Section 7.1.1, and forwards the resulting second ClientHelloInner. The client-facing server forwards all other TLS messages between the client and backend server unmodified.

Otherwise, if all candidate ECHConfigs fail to decrypt the extension, the client-facing server MUST ignore the extension and proceed with the connection using ClientHelloOuter. This connection proceeds as usual, except the server MUST include the "encrypted_client_hello" extension in its EncryptedExtensions with the "retry_configs" field set to one or more ECHConfig structures with up-to-date keys. Servers MAY supply multiple ECHConfig values of different versions. This allows a server to support multiple versions at once.

Note that decryption failure could indicate a GREASE ECH extension (see Section 6.4), so it is necessary for servers to proceed with the connection and rely on the client to abort if ECH was required. In particular, the unrecognized value alone does not indicate a misconfigured ECH advertisement (Section 8.1). Instead, servers can measure occurrences of the "ech_required" alert to detect this case.

7.1.1. HelloRetryRequest

In case a HelloRetryRequest (HRR) is sent, the client-facing server MUST consistently accept or decline ECH between the two ClientHellos, using the same ECHConfig, and abort the handshake if this is not possible. This is achieved as follows. Let CH1 and CH2 denote, respectively, the first and second ClientHello transmitted on the wire by the client:

1. If CH1 contains the "encrypted_client_hello" extension but CH2 does not, or if CH2 contains the "encrypted_client_hello" extension but CH1 does not, then the server MUST abort the handshake with an "illegal_parameter" alert.

2. If the "encrypted_client_hello" extension is sent in CH2, the server follows the procedure in Section 7.1 to decrypt the extension, but it uses the previously-selected ECHConfig as the set of candidate ECHConfigs. If decryption fails, the server aborts the connection with a "decrypt_error" alert rather than continuing the handshake with the second ClientHelloOuter.

[[OPEN ISSUE: If the client-facing server implements stateless HRR, it has no way to send a cookie, short of as-yet-unspecified integration with the backend server. Stateful HRR on the client-facing server works fine, however. See issue #333.]]

7.2. Backend Server Behavior

When the client-facing server accepts ECH, it forwards the ClientHelloInner to the backend server, who terminates the connection. If the ClientHelloInner contains an empty "encrypted_client_hello" extension, then the backend server MUST confirm ECH acceptance by setting ServerHello.random[24:32] to

```
accept_confirmation = HKDF-Expand-Label(
    HKDF-Extract(0, ClientHelloInner.random),
    "ech accept confirmation",
    ServerHello.random[0:24], 8)
```

where HKDF-Expand-Label and HKDF-Extract are as defined in [RFC8446]. The value of ServerHello.random[0:24] is generated as usual by invoking a secure random number generator (see [RFC8446], Section 4.1.2).

8. Compatibility Issues

Unlike most TLS extensions, placing the SNI value in an ECH extension is not interoperable with existing servers, which expect the value in the existing plaintext extension. Thus server operators SHOULD ensure servers understand a given set of ECH keys before advertising them. Additionally, servers SHOULD retain support for any previously-advertised keys for the duration of their validity

However, in more complex deployment scenarios, this may be difficult to fully guarantee. Thus this protocol was designed to be robust in case of inconsistencies between systems that advertise ECH keys and servers, at the cost of extra round-trips due to a retry. Two specific scenarios are detailed below.

8.1. Misconfiguration and Deployment Concerns

It is possible for ECH advertisements and servers to become inconsistent. This may occur, for instance, from DNS misconfiguration, caching issues, or an incomplete rollout in a multi-server deployment. This may also occur if a server loses its ECH keys, or if a deployment of ECH must be rolled back on the server.

The retry mechanism repairs inconsistencies, provided the server is authoritative for the public name. If server and advertised keys mismatch, the server will respond with ech_retry_requested. If the server does not understand the "encrypted_client_hello" extension at all, it will ignore it as required by [RFC8446]; Section 4.1.2. Provided the server can present a certificate valid for the public name, the client can safely retry with updated settings, as described in Section 6.3.

Unless ECH is disabled as a result of successfully establishing a connection to the public name, the client MUST NOT fall back to using unencrypted ClientHellos, as this allows a network attacker to disclose the contents of this ClientHello, including the SNI. It MAY attempt to use another server from the DNS results, if one is provided.

8.2. Middleboxes

A more serious problem is MITM proxies which do not support this extension. [RFC8446], Section 9.3 requires that such proxies remove any extensions they do not understand. The handshake will then present a certificate based on the public name, without echoing the "encrypted_client_hello" extension to the client.

Depending on whether the client is configured to accept the proxy's certificate as authoritative for the public name, this may trigger the retry logic described in Section 6.3 or result in a connection failure. A proxy which is not authoritative for the public name cannot forge a signal to disable ECH.

A non-conformant MITM proxy which instead forwards the ECH extension, substituting its own KeyShare value, will result in the client-facing server recognizing the key, but failing to decrypt the SNI. This causes a hard failure. Clients SHOULD NOT attempt to repair the connection in this case.

9. Compliance Requirements

In the absence of an application profile standard specifying otherwise, a compliant ECH application MUST implement the following HPKE cipher suite:

* KEM: DHKEM(X25519, HKDF-SHA256) (see [I-D.irtf-cfrg-hpke], Section 7.1)

* KDF: HKDF-SHA256 (see [I-D.irtf-cfrg-hpke], Section 7.2)

* AEAD: AES-128-GCM (see [I-D.irtf-cfrg-hpke], Section 7.3)

10. Security Considerations

10.1. Security and Privacy Goals

ECH considers two types of attackers: passive and active. Passive attackers can read packets from the network. They cannot perform any sort of active behavior such as probing servers or querying DNS. A middlebox that filters based on plaintext packet contents is one example of a passive attacker. In contrast, active attackers can write packets into the network for malicious purposes, such as interfering with existing connections, probing servers, and querying DNS. In short, an active attacker corresponds to the conventional threat model for TLS 1.3 [RFC8446].

Given these types of attackers, the primary goals of ECH are as follows.

1. Use of ECH does not weaken the security properties of TLS without ECH.

2. TLS connection establishment to a host with a specific ECHConfig and TLS configuration is indistinguishable from a connection to any other host with the same ECHConfig and TLS configuration. (The set of hosts which share the same ECHConfig and TLS configuration is referred to as the anonymity set.)

Client-facing server configuration determines the size of the anonymity set. For example, if a client-facing server uses distinct ECHConfig values for each host, then each anonymity set has size k = 1. Client-facing servers SHOULD deploy ECH in such a way so as to maximize the size of the anonymity set where possible. This means client-facing servers should use the same ECHConfig for as many hosts as possible. An attacker can distinguish two hosts that have different ECHConfig values based on the ClientECH.config_id value. This also means public information in a TLS handshake is also consistent across hosts. For example, if a client-facing server services many backend origin hosts, only one of which supports some cipher suite, it may be possible to identify that host based on the contents of unencrypted handshake messages.

Beyond these primary security and privacy goals, ECH also aims to hide, to some extent, (a) whether or not a specific server supports ECH and (b) whether or not ECH was accepted for a particular connection. ECH aims to achieve both properties, assuming the attacker is passive and does not know the set of ECH configurations offered by the client-facing server. It does not achieve these properties for active attackers. More specifically:

* Passive attackers with a known ECH configuration can distinguish between a connection that negotiates ECH with that configuration and one which does not, because the latter used a GREASE "encrypted_client_hello" extension (as specified in Section 6.4) or a different ECH configuration.

* Passive attackers without the ECH configuration cannot distinguish between a connection that negotiates ECH and one which uses a GREASE "encrypted_client_hello" extension.

* Active attackers can distinguish between a connection that negotiates ECH and one which uses a GREASE "encrypted_client_hello" extension.

See Section 10.8.4 for more discussion about the "do not stick out" criteria from [RFC8744].

10.2. Unauthenticated and Plaintext DNS

In comparison to [I-D.kazuho-protected-sni], wherein DNS Resource Records are signed via a server private key, ECH records have no authenticity or provenance information. This means that any attacker which can inject DNS responses or poison DNS caches, which is a common scenario in client access networks, can supply clients with fake ECH records (so that the client encrypts data to them) or strip the ECH record from the response. However, in the face of an attacker that controls DNS, no encryption scheme can work because the attacker can replace the IP address, thus blocking client connections, or substituting a unique IP address which is 1:1 with the DNS name that was looked up (modulo DNS wildcards). Thus, allowing the ECH records in the clear does not make the situation significantly worse.

Clearly, DNSSEC (if the client validates and hard fails) is a defense against this form of attack, but DoH/DPRIVE are also defenses against DNS attacks by attackers on the local network, which is a common case where ClientHello and SNI encryption are desired. Moreover, as noted in the introduction, SNI encryption is less useful without encryption of DNS queries in transit via DoH or DPRIVE mechanisms.

10.3. Client Tracking

A malicious client-facing server could distribute unique, per-client ECHConfig structures as a way of tracking clients across subsequent connections. On-path adversaries which know about these unique keys could also track clients in this way by observing TLS connection attempts.

The cost of this type of attack scales linearly with the desired number of target clients. Moreover, DNS caching behavior makes targeting individual users for extended periods of time, e.g., using per-client ECHConfig structures delivered via HTTPS RRs with high TTLs, challenging. Clients can help mitigate this problem by flushing any DNS or ECHConfig state upon changing networks.

10.4. Optional Configuration Identifiers and Trial Decryption

Optional configuration identifiers may be useful in scenarios where clients and client-facing servers do not want to reveal information about the client-facing server in the "encrypted_client_hello" extension. In such settings, clients send either an empty config_id or a randomly generated config_id in the ClientECH. (The precise implementation choice for this mechanism is out of scope for this document.) Servers in these settings must perform trial decryption since they cannot identify the client's chosen ECH key using the config_id value. As a result, support for optional configuration identifiers may exacerbate DoS attacks. Specifically, an adversary may send malicious ClientHello messages, i.e., those which will not decrypt with any known ECH key, in order to force wasteful decryption. Servers that support this feature should, for example, implement some form of rate limiting mechanism to limit the damage caused by such attacks.

10.5. Outer ClientHello

Any information that the client includes in the ClientHelloOuter is visible to passive observers. The client SHOULD NOT send values in the ClientHelloOuter which would reveal a sensitive ClientHelloInner property, such as the true server name. It MAY send values associated with the public name in the ClientHelloOuter.

In particular, some extensions require the client send a server-name-specific value in the ClientHello. These values may reveal information about the true server name. For example, the "cached_info" ClientHello extension [RFC7924] can contain the hash of a previously observed server certificate. The client SHOULD NOT send values associated with the true server name in the ClientHelloOuter. It MAY send such values in the ClientHelloInner.

A client may also use different preferences in different contexts. For example, it may send a different ALPN lists to different servers or in different application contexts. A client that treats this context as sensitive SHOULD NOT send context-specific values in ClientHelloOuter.

Values which are independent of the true server name, or other information the client wishes to protect, MAY be included in ClientHelloOuter. If they match the corresponding ClientHelloInner, they MAY be compressed as described in Section 5.1. However, note the payload length reveals information about which extensions are compressed, so inner extensions which only sometimes match the corresponding outer extension SHOULD NOT be compressed.

Clients MAY include additional extensions in ClientHelloOuter to avoid signaling unusual behavior to passive observers, provided the choice of value and value itself are not sensitive. See Section 10.8.4.

10.6. Related Privacy Leaks

ECH requires encrypted DNS to be an effective privacy protection mechanism. However, verifying the server's identity from the Certificate message, particularly when using the X509 CertificateType, may result in additional network traffic that may reveal the server identity. Examples of this traffic may include requests for revocation information, such as OCSP or CRL traffic, or requests for repository information, such as authorityInformationAccess. It may also include implementation-specific traffic for additional information sources as part of verification.

Implementations SHOULD avoid leaking information that may identify the server. Even when sent over an encrypted transport, such requests may result in indirect exposure of the server's identity, such as indicating a specific CA or service being used. To mitigate this risk, servers SHOULD deliver such information in-band when possible, such as through the use of OCSP stapling, and clients SHOULD take steps to minimize or protect such requests during certificate validation.

10.7. Attacks Exploiting Acceptance Confirmation

To signal acceptance, the backend server overwrites 8 bytes of its ServerHello.random with a value derived from the ClientHelloInner.random. (See Section 7.2 for details.) This behavior increases the likelihood of the ServerHello.random colliding with the ServerHello.random of a previous session, potentially reducing the overall security of the protocol. However, the remaining 24 bytes provide enough entropy to ensure this is not a practical avenue of attack.

On the other hand, the probability that two 8-byte strings are the same is non-negligible. This poses a modest operational risk. Suppose the client-facing server terminates the connection (i.e., ECH is rejected or bypassed): if the last 8 bytes of its ServerHello.random coincide with the confirmation signal, then the client will incorrectly presume acceptance and proceed as if the backend server terminated the connection. However, the probability of a false positive occurring for a given connection is only 1 in 2^64. This value is smaller than the probability of network connection failures in practice.

Note that the same bytes of the ServerHello.random are used to implement downgrade protection for TLS 1.3 (see [RFC8446], Section 4.1.3). The backend server's signal of acceptance does not interfere with this mechanism because ECH is only supported in TLS 1.3 or higher.

10.8. Comparison Against Criteria

[RFC8744] lists several requirements for SNI encryption. In this section, we re-iterate these requirements and assess the ECH design against them.

10.8.1. Mitigate Cut-and-Paste Attacks

Since servers process either ClientHelloInner or ClientHelloOuter, and because ClientHelloInner.random is encrypted, it is not possible for an attacker to "cut and paste" the ECH value in a different Client Hello and learn information from ClientHelloInner.

10.8.2. Avoid Widely Shared Secrets

This design depends upon DNS as a vehicle for semi-static public key distribution. Server operators may partition their private keys however they see fit provided each server behind an IP address has the corresponding private key to decrypt a key. Thus, when one ECH key is provided, sharing is optimally bound by the number of hosts that share an IP address. Server operators may further limit sharing by publishing different DNS records containing ECHConfig values with different keys using a short TTL.

10.8.3. Prevent SNI-Based Denial-of-Service Attacks

This design requires servers to decrypt ClientHello messages with ClientECH extensions carrying valid digests. Thus, it is possible for an attacker to force decryption operations on the server. This attack is bound by the number of valid TCP connections an attacker can open.

10.8.4. Do Not Stick Out

The only explicit signal indicating possible use of ECH is the ClientHello "encrypted_client_hello" extension. Server handshake messages do not contain any signal indicating use or negotiation of ECH. Clients MAY GREASE the "encrypted_client_hello" extension, as described in Section 6.4, which helps ensure the ecosystem handles ECH correctly. Moreover, as more clients enable ECH support, e.g., as normal part of Web browser functionality, with keys supplied by shared hosting providers, the presence of ECH extensions becomes less unusual and part of typical client behavior. In other words, if all Web browsers start using ECH, the presence of this value will not signal unusual behavior to passive eavesdroppers.

10.8.5. Maintain Forward Secrecy

This design is not forward secret because the server's ECH key is static. However, the window of exposure is bound by the key lifetime. It is RECOMMENDED that servers rotate keys frequently.

10.8.6. Enable Multi-party Security Contexts

This design permits servers operating in Split Mode to forward connections directly to backend origin servers. The client authenticates the identity of the backend origin server, thereby avoiding unnecessary MiTM attacks.

Conversely, assuming ECH records retrieved from DNS are authenticated, e.g., via DNSSEC or fetched from a trusted Recursive Resolver, spoofing a client-facing server operating in Split Mode is not possible. See Section 10.2 for more details regarding plaintext DNS.

Authenticating the ECHConfigs structure naturally authenticates the included public name. This also authenticates any retry signals from the client-facing server because the client validates the server certificate against the public name before retrying.

10.8.7. Support Multiple Protocols

This design has no impact on application layer protocol negotiation. It may affect connection routing, server certificate selection, and client certificate verification. Thus, it is compatible with multiple application and transport protocols. By encrypting the entire ClientHello, this design additionally supports encrypting the ALPN extension.

10.9. Padding Policy

Variations in the length of the ClientHelloInner ciphertext could leak information about the corresponding plaintext. Section 6.2 describes a RECOMMENDED padding mechanism for clients aimed at reducing potential information leakage.

10.10. Active Attack Mitigations

This section describes the rationale for ECH properties and mechanics as defenses against active attacks. In all the attacks below, the attacker is on-path between the target client and server. The goal of the attacker is to learn private information about the inner ClientHello, such as the true SNI value.

10.10.1. Client Reaction Attack Mitigation

This attack uses the client's reaction to an incorrect certificate as an oracle. The attacker intercepts a legitimate ClientHello and replies with a ServerHello, Certificate, CertificateVerify, and Finished messages, wherein the Certificate message contains a "test" certificate for the domain name it wishes to query. If the client decrypted the Certificate and failed verification (or leaked information about its verification process by a timing side channel), the attacker learns that its test certificate name was incorrect. As an example, suppose the client's SNI value in its inner ClientHello is "example.com," and the attacker replied with a Certificate for "test.com". If the client produces a verification failure alert because of the mismatch faster than it would due to the Certificate signature validation, information about the name leaks. Note that the attacker can also withhold the CertificateVerify message. In that scenario, a client which first verifies the Certificate would then respond similarly and leak the same information.

```
 Client                         Attacker               Server
   ClientHello
   + key_share
   + ech         ------>      (intercept)     -----> X (drop)

                             ServerHello
                             + key_share
                   {EncryptedExtensions}
                   {CertificateRequest*}
                          {Certificate*}
                    {CertificateVerify*}
                 <------
   Alert
                 ------>
```

Figure 3: Client reaction attack

ClientHelloInner.random prevents this attack. In particular, since the attacker does not have access to this value, it cannot produce the right transcript and handshake keys needed for encrypting the Certificate message. Thus, the client will fail to decrypt the Certificate and abort the connection.

10.10.2. HelloRetryRequest Hijack Mitigation

This attack aims to exploit server HRR state management to recover information about a legitimate ClientHello using its own attacker-controlled ClientHello. To begin, the attacker intercepts and forwards a legitimate ClientHello with an "encrypted_client_hello" (ech) extension to the server, which triggers a legitimate HelloRetryRequest in return. Rather than forward the retry to the client, the attacker, attempts to generate its own ClientHello in response based on the contents of the first ClientHello and HelloRetryRequest exchange with the result that the server encrypts the Certificate to the attacker. If the server used the SNI from the first ClientHello and the key share from the second (attacker-controlled) ClientHello, the Certificate produced would leak the client’s chosen SNI to the attacker.

```
    Client                          Attacker                  Server
      ClientHello
      + key_share
      + ech         ------>       (forward)        ------->
                                                 HelloRetryRequest
                                                       + key_share
                                 (intercept)       <-------

                                 ClientHello
                                 + key_share'
                                 + ech'           ------->
                                                       ServerHello
                                                       + key_share
                                             {EncryptedExtensions}
                                             {CertificateRequest*}
                                                    {Certificate*}
                                              {CertificateVerify*}
                                                        {Finished}
                                                   <-------
                            (process server flight)
```

Figure 4: HelloRetryRequest hijack attack

This attack is mitigated by binding the first and second ClientHello messages together. In particular, since the attacker does not possess the ech_hrr_key, it cannot generate a valid encryption of the second inner ClientHello. The server will attempt decryption using ech_hrr_key, detect failure, and fail the connection.

If the second ClientHello were not bound to the first, it might be possible for the server to act as an oracle if it required parameters from the first ClientHello to match that of the second ClientHello. For example, imagine the client's original SNI value in the inner ClientHello is "example.com", and the attacker's hijacked SNI value in its inner ClientHello is "test.com". A server which checks these for equality and changes behavior based on the result can be used as an oracle to learn the client's SNI.

10.10.3. ClientHello Malleability Mitigation

This attack aims to leak information about secret parts of the encrypted ClientHello by adding attacker-controlled parameters and observing the server's response. In particular, the compression mechanism described in Section 5.1 references parts of a potentially attacker-controlled ClientHelloOuter to construct ClientHelloInner, or a buggy server may incorrectly apply parameters from ClientHelloOuter to the handshake.

To begin, the attacker first interacts with a server to obtain a resumption ticket for a given test domain, such as "example.com". Later, upon receipt of a ClientHelloOuter, it modifies it such that the server will process the resumption ticket with ClientHelloInner. If the server only accepts resumption PSKs that match the server name, it will fail the PSK binder check with an alert when ClientHelloInner is for "example.com" but silently ignore the PSK and continue when ClientHelloInner is for any other name. This introduces an oracle for testing encrypted SNI values.

```
       Client              Attacker                       Server

                                     handshake and ticket
                                        for "example.com"
                                          <-------->

       ClientHello
       + key_share
       + ech
         + outer_extensions(pre_shared_key)
       + pre_shared_key
                    -------->
                           (intercept)
                           ClientHello
                           + key_share
                           + ech
                             + outer_extensions(pre_shared_key)
                           + pre_shared_key'
                                            -------->
                                                          Alert
                                                           -or-
                                                    ServerHello
                                                            ...
                                                       Finished
                                            <--------
```

Figure 5: Message flow for malleable ClientHello

This attack may be generalized to any parameter which the server varies by server name, such as ALPN preferences.

ECH mitigates this attack by only negotiating TLS parameters from ClientHelloInner and authenticating all inputs to the ClientHelloInner (EncodedClientHelloInner and ClientHelloOuter) with the HPKE AEAD. See Section 5.2. An earlier iteration of this specification only encrypted and authenticated the "server_name" extension, which left the overall ClientHello vulnerable to an analogue of this attack.

11. IANA Considerations

11.1. Update of the TLS ExtensionType Registry

IANA is requested to create the following two entries in the existing registry for ExtensionType (defined in [RFC8446]):

1. encrypted_client_hello(0xfe08), with "TLS 1.3" column values being set to "CH, EE", and "Recommended" column being set to "Yes".

2. outer_extensions(0xfd00), with the "TLS 1.3" column values being set to "", and "Recommended" column being set to "Yes".

11.2. Update of the TLS Alert Registry

IANA is requested to create an entry, ech_required(121) in the existing registry for Alerts (defined in [RFC8446]), with the "DTLS-OK" column being set to "Y".

12. ECHConfig Extension Guidance

Any future information or hints that influence ClientHelloOuter SHOULD be specified as ECHConfig extensions. This is primarily because the outer ClientHello exists only in support of ECH. Namely, it is both an envelope for the encrypted inner ClientHello and enabler for authenticated key mismatch signals (see Section 7). In contrast, the inner ClientHello is the true ClientHello used upon ECH negotiation.

13. References

13.1. Normative References

[HTTPS-RR] Schwartz, B., Bishop, M., and E. Nygren, "Service binding and parameter specification via the DNS (DNS SVCB and HTTPS RRs)", Work in Progress, Internet-Draft, draft-ietf-dnsop-svcb-https-01, 13 July 2020, <http://www.ietf.org/internet-drafts/draft-ietf-dnsop-svcb-https-01.txt>.

[I-D.ietf-tls-exported-authenticator]
Sullivan, N., "Exported Authenticators in TLS", Work in Progress, Internet-Draft, draft-ietf-tls-exported-authenticator-13, 26 June 2020, <http://www.ietf.org/internet-drafts/draft-ietf-tls-exported-authenticator-13.txt>.

[I-D.irtf-cfrg-hpke]
Barnes, R., Bhargavan, K., Lipp, B., and C. Wood, "Hybrid Public Key Encryption", Work in Progress, Internet-Draft, draft-irtf-cfrg-hpke-05, 30 July 2020, <http://www.ietf.org/internet-drafts/draft-irtf-cfrg-hpke-05.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>.

[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>.

[RFC7918] Langley, A., Modadugu, N., and B. Moeller, "Transport Layer Security (TLS) False Start", RFC 7918, DOI 10.17487/RFC7918, August 2016, <https://www.rfc-editor.org/info/rfc7918>.

[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>.

[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>.

13.2. Informative References

[I-D.kazuho-protected-sni]
Oku, K., "TLS Extensions for Protecting SNI", Work in Progress, Internet-Draft, draft-kazuho-protected-sni-00, 18 July 2017, <http://www.ietf.org/internet-drafts/draft-kazuho-protected-sni-00.txt>.

[RFC7301] Friedl, S., Popov, A., Langley, A., and E. Stephan, "Transport Layer Security (TLS) Application-Layer Protocol Negotiation Extension", RFC 7301, DOI 10.17487/RFC7301, July 2014, <https://www.rfc-editor.org/info/rfc7301>.

[RFC7858] Hu, Z., Zhu, L., Heidemann, J., Mankin, A., Wessels, D., and P. Hoffman, "Specification for DNS over Transport Layer Security (TLS)", RFC 7858, DOI 10.17487/RFC7858, May 2016, <https://www.rfc-editor.org/info/rfc7858>.

[RFC7924] Santesson, S. and H. Tschofenig, "Transport Layer Security (TLS) Cached Information Extension", RFC 7924, DOI 10.17487/RFC7924, July 2016, <https://www.rfc-editor.org/info/rfc7924>.

[RFC8094] Reddy, T., Wing, D., and P. Patil, "DNS over Datagram Transport Layer Security (DTLS)", RFC 8094, DOI 10.17487/RFC8094, February 2017, <https://www.rfc-editor.org/info/rfc8094>.

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

[RFC8701] Benjamin, D., "Applying Generate Random Extensions And Sustain Extensibility (GREASE) to TLS Extensibility", RFC 8701, DOI 10.17487/RFC8701, January 2020, <https://www.rfc-editor.org/info/rfc8701>.

[RFC8744] Huitema, C., "Issues and Requirements for Server Name Identification (SNI) Encryption in TLS", RFC 8744, DOI 10.17487/RFC8744, July 2020, <https://www.rfc-editor.org/info/rfc8744>.

Appendix A. Alternative SNI Protection Designs

Alternative approaches to encrypted SNI may be implemented at the TLS or application layer. In this section we describe several alternatives and discuss drawbacks in comparison to the design in this document.

A.1. TLS-layer

A.1.1. TLS in Early Data

In this variant, TLS Client Hellos are tunneled within early data payloads belonging to outer TLS connections established with the client-facing server. This requires clients to have established a previous session --- and obtained PSKs --- with the server. The client-facing server decrypts early data payloads to uncover Client Hellos destined for the backend server, and forwards them onwards as necessary. Afterwards, all records to and from backend servers are forwarded by the client-facing server - unmodified. This avoids double encryption of TLS records.

Problems with this approach are: (1) servers may not always be able to distinguish inner Client Hellos from legitimate application data, (2) nested 0-RTT data may not function correctly, (3) 0-RTT data may not be supported - especially under DoS - leading to availability concerns, and (4) clients must bootstrap tunnels (sessions), costing an additional round trip and potentially revealing the SNI during the initial connection. In contrast, encrypted SNI protects the SNI in a distinct Client Hello extension and neither abuses early data nor requires a bootstrapping connection.

A.1.2. Combined Tickets

In this variant, client-facing and backend servers coordinate to produce "combined tickets" that are consumable by both. Clients offer combined tickets to client-facing servers. The latter parse them to determine the correct backend server to which the Client Hello should be forwarded. This approach is problematic due to non-trivial coordination between client-facing and backend servers for ticket construction and consumption. Moreover, it requires a bootstrapping step similar to that of the previous variant. In contrast, encrypted SNI requires no such coordination.

A.2. Application-layer

A.2.1. HTTP/2 CERTIFICATE Frames

In this variant, clients request secondary certificates with CERTIFICATE_REQUEST HTTP/2 frames after TLS connection completion. In response, servers supply certificates via TLS exported authenticators [I-D.ietf-tls-exported-authenticator] in CERTIFICATE frames. Clients use a generic SNI for the underlying client-facing server TLS connection. Problems with this approach include: (1) one additional round trip before peer authentication, (2) non-trivial application-layer dependencies and interaction, and (3) obtaining the generic SNI to bootstrap the connection. In contrast, encrypted SNI induces no additional round trip and operates below the application layer.

Appendix B. Acknowledgements

This document draws extensively from ideas in [I-D.kazuho-protected-sni], but is a much more limited mechanism because it depends on the DNS for the protection of the ECH key. Richard Barnes, Christian Huitema, Patrick McManus, Matthew Prince, Nick Sullivan, Martin Thomson, and David Benjamin also provided important ideas and contributions.

Authors' Addresses

Eric Rescorla
RTFM, Inc.

Email: ekr@rtfm.com

Kazuho Oku
Fastly

Email: kazuhooku@gmail.com

Nick Sullivan
Cloudflare

Email: nick@cloudflare.com

Christopher A. Wood
Cloudflare

Email: caw@heapingbits.net