Stream: Internet Engineering Task Force (IETF)
RFC: 9849
Category: Standards Track
Published: February 2026
ISSN: 2070-1721
Authors: E. Rescorla *Independent* K. Oku *Fastly* N. Sullivan *Cryptography Consulting LLC* C. A. Wood *Apple*

RFC 9849
TLS Encrypted Client Hello

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 4
2. Conventions and Definitions 5
3. Overview 5
  3.1. Topologies 5
  3.2. Encrypted ClientHello (ECH) 6
4. Encrypted ClientHello Configuration 7
  4.1. Configuration Identifiers 9
  4.2. Configuration Extensions 9
5. The "encrypted_client_hello" Extension 10
  5.1. Encoding the ClientHelloInner 11
  5.2. Authenticating the ClientHelloOuter 13
6. Client Behavior 13
  6.1. Offering ECH 13
    6.1.1. Encrypting the ClientHello 14
    6.1.2. GREASE PSK 15
    6.1.3. Recommended Padding Scheme 16
    6.1.4. Determining ECH Acceptance 16
    6.1.5. Handshaking with ClientHelloInner 17
    6.1.6. Handshaking with ClientHelloOuter 18
    6.1.7. Authenticating for the Public Name 19
    6.1.8. Impact of Retry on Future Connections 19
  6.2. GREASE ECH 20
    6.2.1. Client Greasing 20
    6.2.2. Server Greasing 21
7. Server Behavior 21
  7.1. Client-Facing Server 22
    7.1.1. Sending HelloRetryRequest 24

- 7.2. Backend Server 24
  - 7.2.1. Sending HelloRetryRequest 25
- 8. Deployment Considerations 25
  - 8.1. Compatibility Issues 26
    - 8.1.1. Misconfiguration and Deployment Concerns 26
    - 8.1.2. Middleboxes 26
  - 8.2. Deployment Impact 27
- 9. Compliance Requirements 27
- 10. Security Considerations 27
  - 10.1. Security and Privacy Goals 27
  - 10.2. Unauthenticated and Plaintext DNS 29
  - 10.3. Client Tracking 29
  - 10.4. Ignored Configuration Identifiers and Trial Decryption 29
  - 10.5. Outer ClientHello 30
  - 10.6. Inner ClientHello 30
  - 10.7. Related Privacy Leaks 30
  - 10.8. Cookies 31
  - 10.9. Attacks Exploiting Acceptance Confirmation 31
  - 10.10. Comparison Against Criteria 32
    - 10.10.1. Mitigate Cut-and-Paste Attacks 32
    - 10.10.2. Avoid Widely Shared Secrets 32
    - 10.10.3. SNI-Based Denial-of-Service Attacks 32
    - 10.10.4. Do Not Stick Out 32
    - 10.10.5. Maintain Forward Secrecy 33
    - 10.10.6. Enable Multi-party Security Contexts 33
    - 10.10.7. Support Multiple Protocols 34
  - 10.11. Padding Policy 34
  - 10.12. Active Attack Mitigations 34
    - 10.12.1. Client Reaction Attack Mitigation 34
    - 10.12.2. HelloRetryRequest Hijack Mitigation 35

10.12.3. ClientHello Malleability Mitigation 36
10.12.4. ClientHelloInner Packet Amplification Mitigation 37
11. IANA Considerations 38
11.1. Update of the TLS ExtensionType Registry 38
11.2. Update of the TLS Alert Registry 38
11.3. ECH Configuration Extension Registry 38
12. References 39
12.1. Normative References 39
12.2. Informative References 40
Appendix A. Linear-Time Outer Extension Processing 41
Acknowledgements 42
Authors' Addresses 42

1. Introduction

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 left unencrypted in TLS 1.3.

This document specifies a new TLS extension called Encrypted Client Hello (ECH) that allows clients to encrypt their `ClientHello` to the TLS server. This protects the SNI and other potentially sensitive fields, such as the Application-Layer Protocol Negotiation (ALPN) list [RFC7301]. Co-located servers with consistent externally visible TLS configurations and behavior, including supported versions and cipher suites and how they respond to incoming client connections, form an anonymity set. (Note that implementation-specific choices, such as extension ordering within TLS messages or division of data into record-layer boundaries, can result in different externally visible behavior, even for servers with consistent TLS configurations.) 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. Deployment implications of this feature are discussed in Section 8.

ECH is not in itself sufficient to protect the identity of the server. The target domain may also be visible through other channels, such as plaintext client DNS queries or visible server IP addresses. However, encrypted DNS mechanisms such as DNS over HTTPS [RFC8484], DNS over TLS/DTLS [RFC7858] [RFC8094], and DNS over QUIC [RFC9250] provide mechanisms for clients to conceal DNS lookups from network inspection, and many TLS servers host multiple domains on the same IP address. Private origins may also be deployed behind a common provider, such as a reverse proxy. In such environments, the SNI remains the primary explicit signal available to observers to determine the server's identity.

ECH is supported in TLS 1.3 [RFC8446], DTLS 1.3 [RFC9147], and newer versions of the TLS and DTLS protocols.

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". These modes are described in the following section.

3.1. Topologies

```
                  +---------------------+
                  |                     |
                  |   2001:DB8::1111    |
                  |                     |
Client <----->    | private.example.org |
                  |                     |
                  | public.example.com  |
                  |                     |
                  +---------------------+
                          Server
            (Client-Facing and Backend Combined)
```

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.org |
                |                     |     |                     |
                +---------------------+     +---------------------+
                 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, the service provider does not have access to the plaintext of the connection beyond the unencrypted portions of the handshake.

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.

See Section 10 for more discussion about the ECH threat model and how it relates to the client, client-facing server, and backend server.

3.2. Encrypted ClientHello (ECH)

A client-facing server enables ECH by publishing an ECH configuration, which is an encryption public key and associated metadata. Domains which wish to use ECH must publish this configuration, using the key associated with the client-facing server. This document defines the ECH configuration's format, but delegates DNS publication details to [RFC9460]. See [RFCYYY1] for specifics about how ECH configurations are advertised in SVCB and HTTPS records. Other delivery mechanisms are also possible. For example, the client may have the ECH configuration preconfigured.

When a client wants to establish a TLS session with some backend server, it constructs a private `ClientHello`, referred to as the `ClientHelloInner`. The client then constructs a public `ClientHello`, referred to as the `ClientHelloOuter`. The `ClientHelloOuter` contains innocuous values for sensitive extensions and an "encrypted_client_hello" extension (Section 5), which carries the encrypted `ClientHelloInner`. Finally, the client sends `ClientHelloOuter` to the server.

The server takes one of the following actions:

1. If it does not support ECH or cannot decrypt the extension, it completes the handshake with `ClientHelloOuter`. This is referred to as rejecting ECH.
2. If it successfully decrypts the extension, it forwards the `ClientHelloInner` to the backend server, which completes the handshake. This is referred to as accepting ECH.

Upon receiving the server's response, the client determines whether or not ECH was accepted (Section 6.1.4) and proceeds with the handshake accordingly. When ECH is rejected, the resulting connection is not usable by the client for application data. Instead, ECH rejection allows the client to retry with up-to-date configuration (Section 6.1.6).

The primary goal of ECH is to ensure that connections to servers in the same anonymity set are indistinguishable from one another. Moreover, it should achieve this goal 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 Hybrid Public Key Encryption (HPKE) for public key encryption [HPKE]. The ECH configuration is defined by the following `ECHConfig` structure.

```
    opaque HpkePublicKey<1..2^16-1>;
    uint16 HpkeKemId;              // Defined in RFC 9180
    uint16 HpkeKdfId;              // Defined in RFC 9180
    uint16 HpkeAeadId;             // Defined in RFC 9180
    uint16 ECHConfigExtensionType; // Defined in Section 11.3

    struct {
        HpkeKdfId kdf_id;
        HpkeAeadId aead_id;
    } HpkeSymmetricCipherSuite;

    struct {
        uint8 config_id;
        HpkeKemId kem_id;
        HpkePublicKey public_key;
        HpkeSymmetricCipherSuite cipher_suites<4..2^16-4>;
    } HpkeKeyConfig;

    struct {
        ECHConfigExtensionType type;
        opaque data<0..2^16-1>;
    } ECHConfigExtension;

    struct {
        HpkeKeyConfig key_config;
        uint8 maximum_name_length;
        opaque public_name<1..255>;
        ECHConfigExtension extensions<0..2^16-1>;
    } ECHConfigContents;

    struct {
        uint16 version;
        uint16 length;
        select (ECHConfig.version) {
          case 0xfe0d: ECHConfigContents contents;
        }
    } ECHConfig;
```

The structure contains the following fields:

version: The version of ECH for which this configuration is used. 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. This length field allows implementations to skip over the elements in such a list where they cannot parse the specific version of `ECHConfig`.

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:

key_config: A `HpkeKeyConfig` structure carrying the configuration information associated with the HPKE public key (an "ECH key"). Note that this structure contains the `config_id` field, which applies to the entire `ECHConfigContents`.

`maximum_name_length`: The longest name of a backend server, if known. If not known, this value can be set to zero. It is used to compute padding (Section 6.1.3) and does not constrain server name lengths. Names may exceed this length if, e.g., the server uses wildcard names or added new names to the anonymity set.

public_name: The DNS name of the client-facing server, i.e., the entity trusted to update the ECH configuration. This is used to correct misconfigured clients, as described in Section 6.1.6.

See Section 6.1.7 for how the client interprets and validates the public_name.

extensions: A list of ECHConfigExtension values that the client must take into consideration when generating a `ClientHello` message. Each ECHConfigExtension has a 2-octet type and opaque data value, where the data value is encoded with a 2-octet integer representing the length of the data, in network byte order. ECHConfigExtension values are described below (Section 4.2).

The `HpkeKeyConfig` structure contains the following fields:

`config_id`: A one-byte identifier for the given HPKE key configuration. This is used by clients to indicate the key used for `ClientHello` encryption. Section 4.1 describes how client-facing servers allocate this value.

kem_id: The HPKE Key Encapsulation Mechanism (KEM) identifier corresponding to `public_key`. Clients MUST ignore any `ECHConfig` structure with a key using a KEM they do not support.

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

cipher_suites: The list of HPKE Key Derivation Function (KDF) and Authenticated Encryption with Associated Data (AEAD) identifier pairs clients can use for encrypting `ClientHelloInner`. See Section 6.1 for how clients choose from this list.

The client-facing server advertises a sequence of ECH configurations to clients, serialized as follows.

```
    ECHConfig ECHConfigList<4..2^16-1>;
```

The `ECHConfigList` 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.

4.1. Configuration Identifiers

A client-facing server has a set of known `ECHConfig` values with corresponding private keys. This set SHOULD contain the currently published values, as well as previous values that may still be in use, since clients may cache DNS records up to a TTL or longer.

Section 7.1 describes a trial decryption process for decrypting the `ClientHello`. This can impact performance when the client-facing server maintains many known `ECHConfig` values. To avoid this, the client-facing server SHOULD allocate distinct `config_id` values for each `ECHConfig` in its known set. The RECOMMENDED strategy is via rejection sampling, i.e., to randomly select `config_id` repeatedly until it does not match any known `ECHConfig`.

It is not necessary for `config_id` values across different client-facing servers to be distinct. A backend server may be hosted behind two different client-facing servers with colliding `config_id` values without any performance impact. Values may also be reused if the previous `ECHConfig` is no longer in the known set.

4.2. Configuration Extensions

ECH configuration extensions are used to provide room for additional functionality as needed. The format is as defined in Section 4 and mirrors Section 4.2 of [RFC8446]. However, ECH configuration extension types are maintained by IANA as described in Section 11.3. ECH configuration extensions follow the same interpretation rules as TLS extensions: extensions MAY appear in any order, but there MUST NOT be more than one extension of the same type in the extensions block. Unlike TLS extensions, an extension can be tagged as mandatory by using an extension type codepoint with the high order bit set to 1.

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

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 an enabler for authenticated key mismatch signals (see Section 7). In contrast, the inner `ClientHello` is the true `ClientHello` used upon ECH negotiation.

5. The "encrypted_client_hello" Extension

To offer ECH, the client sends an "encrypted_client_hello" extension in the `ClientHelloOuter`. When it does, it MUST also send the extension in `ClientHelloInner`.

~~ enum { encrypted_client_hello(0xfe0d), (65535) } ExtensionType; ~~

The payload of the extension has the following structure:

```
    enum { outer(0), inner(1) } ECHClientHelloType;

    struct {
       ECHClientHelloType type;
       select (ECHClientHello.type) {
           case outer:
               HpkeSymmetricCipherSuite cipher_suite;
               uint8 config_id;
               opaque enc<0..2^16-1>;
               opaque payload<1..2^16-1>;
           case inner:
               Empty;
       };
    } ECHClientHello;
```

The outer extension uses the `outer` variant and the inner extension uses the `inner` variant. The inner extension has an empty payload, which is included because TLS servers are not allowed to provide extensions in ServerHello which were not included in `ClientHello`. The outer extension has the following fields:

`config_id`: The `ECHConfigContents.key_config.config_id` for the chosen `ECHConfig`.

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

enc: The HPKE encapsulated key used by servers to decrypt the corresponding `payload` field. This field is empty in a `ClientHelloOuter` sent in response to HelloRetryRequest.

payload: The serialized and encrypted `EncodedClientHelloInner` structure, encrypted using HPKE as described in Section 6.1.

When a client offers the `outer` version of an "encrypted_client_hello" extension, the server MAY include an "encrypted_client_hello" extension in its EncryptedExtensions message, as described in Section 7.1, with the following payload:

~~ struct { ECHConfigList retry_configs; } ECHEncryptedExtensions; ~~

The response is valid only when the server used the `ClientHelloOuter`. If the server sent this extension in response to the `inner` variant, then the client MUST abort with an "unsupported_extension" alert.

retry_configs: An `ECHConfigList` structure containing one or more `ECHConfig` structures, in decreasing order of preference, to be used by the client as described in Section 6.1.6. These are known as the server's "retry configurations".

Finally, when the client offers the "encrypted_client_hello", if the payload is the `inner` variant and the server responds with HelloRetryRequest, it MUST include an "encrypted_client_hello" extension with the following payload:

~~ struct { opaque confirmation[8]; } ECHHelloRetryRequest; ~~

The value of ECHHelloRetryRequest.confirmation is set to `hrr_accept_confirmation` as described in Section 7.2.1.

This document also defines the "ech_required" alert, which the client 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

Before encrypting, the client pads and optionally compresses `ClientHelloInner` into an `EncodedClientHelloInner` structure, defined below:

~~ struct { ClientHello client_hello; uint8 zeros[length_of_padding]; } EncodedClientHelloInner; ~~

The `client_hello` field is computed by first making a copy of `ClientHelloInner` and setting the `legacy_session_id` field to the empty string. In TLS, this field uses the `ClientHello` structure defined in Section 4.1.2 of [RFC8446]. In DTLS, it uses the `ClientHello` structure defined in Section 5.3 of [RFC9147]. This does not include Handshake structure's four-byte header in TLS, nor twelve-byte header in DTLS. The `zeros` field MUST be all zeroes of length `length_of_padding` (see Section 6.1.3).

Repeating large extensions, such as "key_share" with post-quantum algorithms, between `ClientHelloInner` and `ClientHelloOuter` can lead to excessive size. To reduce the size impact, the client MAY substitute extensions which it knows will be duplicated in `ClientHelloOuter`. It does so by removing and replacing extensions from `EncodedClientHelloInner` with a single "ech_outer_extensions" extension, defined as follows:

~~ enum { ech_outer_extensions(0xfd00), (65535) } ExtensionType;

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

OuterExtensions contains the removed ExtensionType values. Each value references the matching extension in `ClientHelloOuter`. The values MUST be ordered contiguously in `ClientHelloInner`, and the "ech_outer_extensions" extension MUST be inserted in the corresponding position in `EncodedClientHelloInner`. Additionally, the extensions MUST appear in `ClientHelloOuter` in the same relative order. However, there is no requirement that they be contiguous. For example, OuterExtensions may contain extensions A, B, and C, while `ClientHelloOuter` contains extensions A, D, B, C, E, and F.

The "ech_outer_extensions" extension can only be included in `EncodedClientHelloInner` and MUST NOT appear in either `ClientHelloOuter` or `ClientHelloInner`.

Finally, the client pads the message by setting the `zeros` field to a byte string whose contents are all zeros and whose length is the amount of padding to add. Section 6.1.3 describes a recommended padding scheme.

The client-facing server computes `ClientHelloInner` by reversing this process. First, it parses `EncodedClientHelloInner`, interpreting all bytes after `client_hello` as padding. If any padding byte is non-zero, the server MUST abort the connection with an "illegal_parameter" alert.

Next, it makes a copy of the `client_hello` field and copies the `legacy_session_id` field from `ClientHelloOuter`. It then looks for an "ech_outer_extensions" extension. If found, it replaces the extension with the corresponding sequence of extensions in the `ClientHelloOuter`. The server MUST abort the connection with an "illegal_parameter" alert if any of the following are true:

- Any referenced extension is missing in `ClientHelloOuter`.
- Any extension is referenced in OuterExtensions more than once.
- "encrypted_client_hello" is referenced in OuterExtensions.
- The extensions in `ClientHelloOuter` corresponding to those in OuterExtensions do not occur in the same order.

These requirements prevent an attacker from performing a packet amplification attack by crafting a `ClientHelloOuter` which decompresses to a much larger `ClientHelloInner`. This is discussed further in Section 10.12.4.

Implementations SHOULD construct the `ClientHelloInner` in linear time. Quadratic time implementations (such as may happen via naive copying) create a denial-of-service risk. Appendix A describes a linear-time procedure that may be used for this purpose.

5.2. Authenticating the ClientHelloOuter

To prevent a network attacker from modifying the `ClientHelloOuter` while keeping the same encrypted `ClientHelloInner` (see Section 10.12.3), ECH authenticates `ClientHelloOuter` by passing `ClientHelloOuterAAD` as the associated data for HPKE sealing and opening operations. The `ClientHelloOuterAAD` is a serialized `ClientHello` structure, defined in Section 4.1.2 of [RFC8446] for TLS and Section 5.3 of [RFC9147] for DTLS, which matches the `ClientHelloOuter` except that the `payload` field of the "encrypted_client_hello" is replaced with a byte string of the same length but whose contents are zeros. This value does not include Handshake structure's four-byte header in TLS nor twelve-byte header in DTLS.

6. Client Behavior

Clients that implement the ECH extension behave in one of two ways: either they offer a real ECH extension, as described in Section 6.1, or they send a Generate Random Extensions And Sustain Extensibility (GREASE) [RFC8701] ECH extension, as described in Section 6.2. Clients of the latter type do not negotiate ECH. Instead, they generate a dummy ECH extension that is ignored by the server. (See Section 10.10.4 for an explanation.) The client offers ECH if it is in possession of a compatible ECH configuration and sends GREASE ECH (see Section 6.2) otherwise.

6.1. Offering ECH

To offer ECH, the client first chooses a suitable `ECHConfig` from the server's `ECHConfigList`. To determine if a given `ECHConfig` is suitable, it checks that it supports the KEM algorithm identified by `ECHConfig.contents.kem_id`, at least one KDF/AEAD algorithm identified by `ECHConfig.contents.cipher_suites`, and the version of ECH indicated by `ECHConfig.version`. Once a suitable configuration is found, the client selects the cipher suite it will use for encryption. It MUST NOT choose a cipher suite or version not advertised by the configuration. If no compatible configuration is found, then the client SHOULD proceed as described in Section 6.2.

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. 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. If it intends to compress any extensions (see Section 5.1), it MUST order those extensions consecutively.
4. It MUST include the "encrypted_client_hello" extension of type `inner` as described in Section 5. (This requirement is not applicable when the "encrypted_client_hello" extension is generated as described in Section 6.2.)

The client then constructs `EncodedClientHelloInner` as described in Section 5.1. It also computes an HPKE encryption context and `enc` value as:

~~ pkR = DeserializePublicKey(ECHConfig.contents.public_key) enc, context = SetupBaseS(pkR, "tls ech" || 0x00 || ECHConfig) ~~

Next, it constructs a partial `ClientHelloOuterAAD` 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`. The copied extensions additionally MUST be in the same relative order as in `ClientHelloInner`.
3. 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. Note that compatibility mode is not used in DTLS 1.3, but following this rule will produce the correct results for both TLS 1.3 and DTLS 1.3.
4. 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 10.12.1.)
5. It SHOULD place the value of `ECHConfig.contents.public_name` in the "server_name" extension. Clients that do not follow this step, or place a different value in the "server_name" extension, risk breaking the retry mechanism described in Section 6.1.6 or failing to interoperate with servers that require this step to be done; see Section 7.1.
6. When the client offers the "pre_shared_key" extension in `ClientHelloInner`, it SHOULD also include a GREASE "pre_shared_key" extension in `ClientHelloOuter`, generated in the manner described in Section 6.1.2. The client MUST NOT use this extension to advertise a PSK to the client-facing server. (See Section 10.12.3.) When the client includes a GREASE "pre_shared_key" extension, it MUST also copy the "psk_key_exchange_modes" from the `ClientHelloInner` into the `ClientHelloOuter`.
7. When the client offers the "early_data" extension in `ClientHelloInner`, it MUST also include the "early_data" extension in `ClientHelloOuter`. This allows servers that reject ECH and use `ClientHelloOuter` to safely ignore any early data sent by the client per [RFC8446], Section 4.2.10.

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.

Finally, the client encrypts the `EncodedClientHelloInner` with the above values, as described in Section 6.1.1, to construct a `ClientHelloOuter`. It sends this to the server and processes the response as described in Section 6.1.4.

6.1.1. Encrypting the ClientHello

Given an `EncodedClientHelloInner`, an HPKE encryption context and `enc` value, and a partial `ClientHelloOuterAAD`, the client constructs a `ClientHelloOuter` as follows.

First, the client determines the length L of encrypting `EncodedClientHelloInner` with the selected HPKE AEAD. This is typically the sum of the plaintext length and the AEAD tag length. The client then completes the `ClientHelloOuterAAD` with an "encrypted_client_hello" extension. This extension value contains the outer variant of `ECHClientHello` with the following fields:

- `config_id`, the identifier corresponding to the chosen `ECHConfig` structure;
- `cipher_suite`, the client's chosen cipher suite;
- `enc`, as given above; and
- `payload`, a placeholder byte string containing L zeros.

If configuration identifiers (see Section 10.4) are to be ignored, `config_id` SHOULD be set to a randomly generated byte in the first `ClientHelloOuter` and, in the event of a HelloRetryRequest (HRR), MUST be left unchanged for the second `ClientHelloOuter`.

The client serializes this structure to construct the `ClientHelloOuterAAD`. It then computes the final payload as:

~~ final_payload = context.Seal(ClientHelloOuterAAD, EncodedClientHelloInner) ~~

Including `ClientHelloOuterAAD` as the HPKE AAD binds the `ClientHelloOuter` to the `ClientHelloInner`, thus preventing attackers from modifying `ClientHelloOuter` while keeping the same `ClientHelloInner`, as described in Section 10.12.3.

Finally, the client replaces `payload` with `final_payload` to obtain `ClientHelloOuter`. The two values have the same length, so it is not necessary to recompute length prefixes in the serialized structure.

Note this construction requires the "encrypted_client_hello" be computed after all other extensions. This is possible because the `ClientHelloOuter`'s "pre_shared_key" extension is either omitted or uses a random binder (Section 6.1.2).

6.1.2. GREASE PSK

When offering ECH, the client is not permitted to advertise PSK identities in the `ClientHelloOuter`. However, the client can send a "pre_shared_key" extension in the `ClientHelloInner`. In this case, when resuming a session with the client, the backend server sends a "pre_shared_key" extension in its ServerHello. This would appear to a network observer as if the server were sending this extension without solicitation, which would violate the extension rules described in [RFC8446]. When offering a PSK in `ClientHelloInner`, clients SHOULD send a GREASE "pre_shared_key" extension in the `ClientHelloOuter` to make it appear to the network as if the extension were negotiated properly.

The client generates the extension payload by constructing an `OfferedPsks` structure (see [RFC8446], Section 4.2.11) as follows. For each PSK identity advertised in the `ClientHelloInner`, the client generates a random PSK identity with the same length. It also generates a random, 32-bit, unsigned integer to use as the `obfuscated_ticket_age`. Likewise, for each inner PSK binder, the client generates a random string of the same length.

Per the rules of Section 6.1, the server is not permitted to resume a connection in the outer handshake. If ECH is rejected and the client-facing server replies with a "pre_shared_key" extension in its ServerHello, then the client MUST abort the handshake with an "illegal_parameter" alert.

6.1.3. Recommended Padding Scheme

If the `ClientHelloInner` is encrypted without padding, then the length of the `ClientHelloOuter.payload` can leak information about `ClientHelloInner`. In order to prevent this, the `EncodedClientHelloInner` structure has a padding field. This section describes a deterministic mechanism for computing the required amount of padding 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. Clients SHOULD use the `ECHConfig`'s `maximum_name_length` field as follows, where M is the `maximum_name_length` value.

1. If the `ClientHelloInner` contained a "server_name" extension with a name of length D, add max(0, M - D) bytes of padding.
2. If the `ClientHelloInner` did not contain a "server_name" extension (e.g., if the client is connecting to an IP address), add M + 9 bytes of padding. This is the length of a "server_name" extension with an M-byte name.

Finally, the client SHOULD pad the entire message as follows:

1. Let L be the length of the `EncodedClientHelloInner` with all the padding computed so far.
2. Let N = 31 - ((L - 1) % 32) and add N bytes of padding.

This rounds the length of `EncodedClientHelloInner` up to a multiple of 32 bytes, reducing the set of possible lengths across all clients.

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.

6.1.4. Determining ECH Acceptance

As described in Section 7, the server may either accept ECH and use `ClientHelloInner` or reject it and use `ClientHelloOuter`. This is determined by the server's initial message.

If the message does not negotiate TLS 1.3 or higher, the server has rejected ECH. Otherwise, it is either a ServerHello or HelloRetryRequest.

If the message is a ServerHello, the client computes `accept_confirmation` as described in Section 7.2. If this value matches the last 8 bytes of `ServerHello.random`, the server has accepted ECH. Otherwise, it has rejected ECH.

If the message is a HelloRetryRequest, the client checks for the "encrypted_client_hello" extension. If none is found, the server has rejected ECH. Otherwise, if it has a length other than 8, the client aborts the handshake with a "decode_error" alert. Otherwise, the client computes `hrr_accept_confirmation` as described in Section 7.2.1. If this value matches the extension payload, the server has accepted ECH. Otherwise, it has rejected ECH.

If the server accepts ECH, the client handshakes with `ClientHelloInner` as described in Section 6.1.5. Otherwise, the client handshakes with `ClientHelloOuter` as described in Section 6.1.6.

6.1.5. Handshaking with ClientHelloInner

If the server accepts ECH, the client proceeds with the connection as in [RFC8446], with the following modifications:

The client behaves as if it had sent `ClientHelloInner` as the `ClientHello`. That is, it evaluates the handshake using the `ClientHelloInner`'s preferences, and, when computing the transcript hash (Section 4.4.1 of [RFC8446]), it uses `ClientHelloInner` as the first `ClientHello`.

If the server responds with a HelloRetryRequest, the client computes the updated `ClientHello` message as follows:

1. It computes a second `ClientHelloInner` based on the first `ClientHelloInner`, as in Section 4.1.4 of [RFC8446]. The `ClientHelloInner`'s "encrypted_client_hello" extension is left unmodified.
2. It constructs `EncodedClientHelloInner` as described in Section 5.1.
3. It constructs a second partial `ClientHelloOuterAAD` message. This message MUST be syntactically valid. The extensions MAY be copied from the original `ClientHelloOuter` unmodified or omitted. If not sensitive, the client MAY copy updated extensions from the second `ClientHelloInner` for compression.
4. It encrypts `EncodedClientHelloInner` as described in Section 6.1.1, using the second partial `ClientHelloOuterAAD`, to obtain a second `ClientHelloOuter`. It reuses the original HPKE encryption context computed in Section 6.1 and uses the empty string for `enc`.

   The HPKE context maintains a sequence number, so this operation internally uses a fresh nonce for each AEAD operation. Reusing the HPKE context avoids an attack described in Section 10.12.2.

The client then sends the second `ClientHelloOuter` to the server. However, as above, it uses the second `ClientHelloInner` for preferences, and both the `ClientHelloInner` messages for the transcript hash. Additionally, it checks the resulting ServerHello for ECH acceptance as in Section 6.1.4. If the ServerHello does not also indicate ECH acceptance, the client MUST terminate the connection with an "illegal_parameter" alert.

6.1.6. Handshaking with ClientHelloOuter

If the server rejects ECH, the client proceeds with the handshake, authenticating for `ECHConfig.contents.public_name` as described in Section 6.1.7. If authentication or the handshake fails, the client MUST return a failure to the calling application. It MUST NOT use the retry configurations. It MUST NOT treat this as a secure signal to disable ECH.

If the server supplied an "encrypted_client_hello" extension in its EncryptedExtensions message, the client MUST check that it is syntactically valid and the client MUST abort the connection with a "decode_error" alert otherwise. If an earlier TLS version was negotiated, the client MUST NOT enable the False Start optimization [RFC7918] for this handshake. If both authentication and the handshake complete successfully, the client MUST perform the processing described below and then abort the connection with an "ech_required" alert before sending any application data to the server.

If the server provided "retry_configs" and if at least one of the values contains a version supported by the client, the client can regard the ECH configuration as securely replaced by the server. It SHOULD retry the handshake with a new transport connection using the retry configurations supplied by the server.

Clients can implement a new transport connection in a way that best suits their deployment. For example, clients can reuse the same server IP address when establishing the new transport connection or they can choose to use a different IP address if provided with options from DNS. ECH does not mandate any specific implementation choices when establishing this new connection.

The retry configurations are meant to be used for retried connections. Further use of retry configurations could yield a tracking vector. In settings where the client will otherwise already let the server track the client, e.g., because the client will send cookies to the server in parallel connections, using the retry configurations for these parallel connections does not introduce a new tracking vector.

If none of the values provided in "retry_configs" contains a supported version, the server did not supply an "encrypted_client_hello" extension in its EncryptedExtensions message, or an earlier TLS version was negotiated, the client can regard ECH as securely disabled by the server, and it SHOULD retry the handshake with a new transport connection and ECH disabled.

Clients SHOULD NOT accept "retry_config" in response to a connection initiated in response to a "retry_config". Sending a "retry_config" in this situation is a signal that the server is misconfigured, e.g., the server might have multiple inconsistent configurations so that the client reached a node with configuration A in the first connection and a node with configuration B in the second. Note that this guidance does not apply to the cases in the previous paragraph where the server has securely disabled ECH.

If a client does not retry, it MUST report an error to the calling application.

6.1.7. Authenticating for the Public Name

When the server rejects ECH, 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 `ECHConfig.contents.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.

In verifying the client-facing server certificate, the client MUST interpret the public name as a DNS-based reference identity [RFC9525]. Clients that incorporate DNS names and IP addresses into the same syntax (e.g. Section 7.4 of [RFC3986] and [WHATWG-IPV4]) MUST reject names that would be interpreted as IPv4 addresses. Clients that enforce this by checking `ECHConfig.contents.public_name` do not need to repeat the check when processing ECH rejection.

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 6.1.6. This may be implemented, for instance, by reporting a failed connection with a dedicated error code.

Prior to attempting a connection, a client SHOULD validate the `ECHConfig`. Clients SHOULD ignore any `ECHConfig` structure with a public_name that is not a valid host name in preferred name syntax (see Section 2 of [DNS-TERMS]). That is, to be valid, the public_name needs to be a dot-separated sequence of LDH labels, as defined in Section 2.3.1 of [RFC5890], where:

- the sequence does not begin or end with an ASCII dot, and
- all labels are at most 63 octets.

Clients additionally SHOULD ignore the structure if the final LDH label either consists of all ASCII digits (i.e., '0' through '9') or is "0x" or "0X" followed by some, possibly empty, sequence of ASCII hexadecimal digits (i.e., '0' through '9', 'a' through 'f', and 'A' through 'F'). This avoids public_name values that may be interpreted as IPv4 literals.

6.1.8. Impact of Retry on Future Connections

Clients MAY use information learned from a rejected ECH for future connections to avoid repeatedly connecting to the same server and being forced to retry. However, they MUST handle ECH rejection for those connections as if it were a fresh connection, rather than enforcing the single retry limit from Section 6.1.6. The reason for this requirement is that if the server sends a "retry_config" and then immediately rejects the resulting connection, it is most likely misconfigured. However, if the server sends a "retry_config" and then the client tries to use that to connect some time later, it is possible that the server has changed its configuration again and is now trying to recover.

Any persisted information MUST be associated with the `ECHConfig` source used to bootstrap the connection, such as a DNS SVCB ServiceMode record [RFCYYY1]. Clients MUST limit any sharing of persisted ECH-related state to connections that use the same `ECHConfig` source. Otherwise, it might become possible for the client to have the wrong public name for the server, making recovery impossible.

ECHConfigs learned from ECH rejection can be used as a tracking vector. Clients SHOULD impose the same lifetime and scope restrictions that they apply to other server-based tracking vectors such as PSKs.

In general, the safest way for clients to minimize ECH retries is to comply with any freshness rules (e.g., DNS TTLs) imposed by the ECH configuration.

6.2. GREASE ECH

The GREASE ECH mechanism allows a connection between an ECH-capable client and a non-ECH server to appear to use ECH, thus reducing the extent to which ECH connections stick out (see Section 10.10.4).

6.2.1. Client Greasing

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 in the first `ClientHello` as follows:

- Set the `config_id` field to a random byte.
- Set the `cipher_suite` field to a supported HpkeSymmetricCipherSuite. 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 `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 the selected AEAD scheme and L is the size of the `EncodedClientHelloInner` the client would compute when offering ECH, padded according to Section 6.1.3.

If sending a second `ClientHello` in response to a HelloRetryRequest, the client copies the entire "encrypted_client_hello" extension from the first `ClientHello`. The identical value will reveal to an observer that the value of "encrypted_client_hello" was fake, but this only occurs if there is a HelloRetryRequest.

If the server sends an "encrypted_client_hello" extension in either HelloRetryRequest or EncryptedExtensions, 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. It MUST NOT save the "retry_configs" value in EncryptedExtensions.

Offering a GREASE extension is not considered offering an encrypted `ClientHello` for purposes of requirements in Section 6.1. In particular, the client MAY offer to resume sessions established without ECH.

6.2.2. Server Greasing

Section 11.3 describes a set of Reserved extensions which will never be registered. These can be used by servers to "grease" the contents of the ECH configuration, as inspired by [RFC8701]. This helps ensure clients process ECH extensions correctly. When constructing ECH configurations, servers SHOULD randomly select from reserved values with the high-order bit clear. Correctly implemented clients will ignore those extensions.

The reserved values with the high-order bit set are mandatory, as defined in Section 4.2. Servers SHOULD randomly select from these values and include them in extraneous ECH configurations. Correctly implemented clients will ignore these configurations because they do not recognize the mandatory extension. Servers SHOULD ensure that any client using these configurations encounters a warning or error message. This can be accomplished in several ways, including:

- By giving the extraneous configurations distinctive config IDs or public names, and rejecting the TLS connection or inserting an application-level warning message when these are observed.
- By giving the extraneous configurations an invalid public key and a public name not associated with the server so that the initial `ClientHelloOuter` will not be decryptable and the server cannot perform the recovery flow described in Section 6.1.6.

7. Server Behavior

As described in Section 3.1, servers can play two roles, either as the client-facing server or as the backend server. Depending on the server role, the `ECHClientHello` will be different:

- A client-facing server expects an `ECHClientHello.type` of `outer`, and proceeds as described in Section 7.1 to extract a `ClientHelloInner`, if available.
- A backend server expects an `ECHClientHello.type` of `inner`, and proceeds as described in Section 7.2.

In split mode, a client-facing server which receives a `ClientHello` with `ECHClientHello.type` of `inner` MUST abort with an "illegal_parameter" alert. Similarly, in split mode, a backend server which receives a `ClientHello` with `ECHClientHello.type` of `outer` MUST abort with an "illegal_parameter" alert.

In shared mode, a server plays both roles, first decrypting the `ClientHelloOuter` and then using the contents of the `ClientHelloInner`. A shared mode server which receives a `ClientHello` with `ECHClientHello.type` of `inner` MUST abort with an "illegal_parameter" alert, because such a `ClientHello` should never be received directly from the network.

If `ECHClientHello.type` is not a valid `ECHClientHelloType`, then the server MUST abort with an "illegal_parameter" alert.

If the "encrypted_client_hello" is not present, then the server completes the handshake normally, as described in [RFC8446].

7.1. Client-Facing Server

Upon receiving an "encrypted_client_hello" extension in an initial `ClientHello`, 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 `ECHConfig` values. This list is determined by one of the two following methods:

1. Compare `ECHClientHello.config_id` against identifiers of each known `ECHConfig` and select the ones that match, if any, as candidates.
2. Collect all known `ECHConfig` values as candidates, with trial decryption below determining the final selection.

Some uses of ECH, such as local discovery mode, may randomize the `ECHClientHello.config_id` since it can be used as a tracking vector. In such cases, the second method SHOULD be used for matching the `ECHClientHello` to a known `ECHConfig`. See Section 10.4. Unless specified by the application profile or otherwise externally configured, implementations MUST use the first method.

The server then iterates over the candidate `ECHConfig` values, attempting to decrypt the "encrypted_client_hello" extension as follows.

The server verifies that the `ECHConfig` supports the cipher suite indicated by the `ECHClientHello.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 `ECHClientHello.payload`, using the private key skR corresponding to `ECHConfig`, as follows:

~~ context = SetupBaseR(ECHClientHello.enc, skR, "tls ech" || 0x00 || ECHConfig) EncodedClientHelloInner = context.Open(ClientHelloOuterAAD, ECHClientHello.payload) ~~

`ClientHelloOuterAAD` is computed from `ClientHelloOuter` as described in Section 5.2. The `info` parameter to SetupBaseR 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 iterating over the candidate `ECHConfig` values.

Once the server has chosen the correct `ECHConfig`, it MAY verify that the value in the `ClientHelloOuter` "server_name" extension matches the value of `ECHConfig.contents.public_name` and abort with an "illegal_parameter" alert if these do not match. This optional check allows the server to limit ECH connections to only use the public SNI values advertised in its ECHConfigs. The server MUST be careful not to unnecessarily reject connections if the same `ECHConfig` id or keypair is used in multiple ECHConfigs with distinct public names.

Upon determining the `ClientHelloInner`, the client-facing server checks that the message includes a well-formed "encrypted_client_hello" extension of type `inner` and that it does not offer TLS 1.2 or below. If either of these checks fails, the client-facing server MUST abort with an "illegal_parameter" alert.

If these checks succeed, 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 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 `ECHConfig` values fail to decrypt the extension, the client-facing server MUST ignore the extension and proceed with the connection using `ClientHelloOuter` with the following modifications:

- If sending a HelloRetryRequest, the server MAY include an "encrypted_client_hello" extension with a payload of 8 random bytes; see Section 10.10.4 for details.
- If the server is configured with any ECHConfigs, it 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 6.2), 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 8.1.1). Instead, servers can measure occurrences of the "ech_required" alert to detect this case.

7.1.1. Sending HelloRetryRequest

After sending or forwarding a HelloRetryRequest, the client-facing server does not repeat the steps in Section 7.1 with the second `ClientHelloOuter`. Instead, it continues with the `ECHConfig` selection from the first `ClientHelloOuter` as follows:

If the client-facing server accepted ECH, it checks that the second `ClientHelloOuter` also contains the "encrypted_client_hello" extension. If not, it MUST abort the handshake with a "missing_extension" alert. Otherwise, it checks that `ECHClientHello.cipher_suite` and `ECHClientHello.config_id` are unchanged, and that `ECHClientHello.enc` is empty. If not, it MUST abort the handshake with an "illegal_parameter" alert.

Finally, it decrypts the new `ECHClientHello.payload` as a second message with the previous HPKE context:

~~ EncodedClientHelloInner = context.Open(ClientHelloOuterAAD, ECHClientHello.payload) ~~

`ClientHelloOuterAAD` is computed as described in Section 5.2, but using the second `ClientHelloOuter`. If decryption fails, the client-facing server MUST abort the handshake with a "decrypt_error" alert. Otherwise, it reconstructs the second `ClientHelloInner` from the new `EncodedClientHelloInner` as described in Section 5.1, using the second `ClientHelloOuter` for any referenced extensions.

The client-facing server then forwards the resulting `ClientHelloInner` to the backend server. It forwards all subsequent TLS messages between the client and backend server unmodified.

If the client-facing server rejected ECH, or if the first `ClientHello` did not include an "encrypted_client_hello" extension, the client-facing server proceeds with the connection as usual. The server does not decrypt the second `ClientHello`'s `ECHClientHello.payload` value, if there is one. Moreover, if the server is configured with any ECHConfigs, it 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, as described in Section 7.1.

Note that a client-facing server that forwards the first `ClientHello` cannot include its own "cookie" extension if the backend server sends a HelloRetryRequest. This means that the client-facing server either needs to maintain state for such a connection or it needs to coordinate with the backend server to include any information it requires to process the second `ClientHello`.

7.2. Backend Server

Upon receipt of an "encrypted_client_hello" extension of type `inner` in a `ClientHello`, if the backend server negotiates TLS 1.3 or higher, then it MUST confirm ECH acceptance to the client by computing its ServerHello as described here.

The backend server embeds in `ServerHello.random` a string derived from the inner handshake. It begins by computing its ServerHello as usual, except the last 8 bytes of `ServerHello.random` are set to zero. It then computes the transcript hash for `ClientHelloInner` up to and including the modified ServerHello, as described in [RFC8446], Section 4.4.1. Let transcript_ech_conf denote the output. Finally, the backend server overwrites the last 8 bytes of the `ServerHello.random` with the following string:

~~ accept_confirmation = HKDF-Expand-Label(HKDF-Extract(0, ClientHelloInner.random), "ech accept confirmation", transcript_ech_conf, 8) ~~

where HKDF-Expand-Label is defined in [RFC8446], Section 7.1, "0" indicates a string of Hash.length bytes set to zero, and Hash is the hash function used to compute the transcript hash. In DTLS, the modified version of HKDF-Expand-Label defined in [RFC9147], Section 5.9 is used instead.

The backend server MUST NOT perform this operation if it negotiated TLS 1.2 or below. Note that doing so would overwrite the downgrade signal for TLS 1.3 (see [RFC8446], Section 4.1.3).

7.2.1. Sending HelloRetryRequest

When the backend server sends HelloRetryRequest in response to the `ClientHello`, it similarly confirms ECH acceptance by adding a confirmation signal to its HelloRetryRequest. But instead of embedding the signal in the HelloRetryRequest.random (the value of which is specified by [RFC8446]), it sends the signal in an extension.

The backend server begins by computing HelloRetryRequest as usual, except that it also contains an "encrypted_client_hello" extension with a payload of 8 zero bytes. It then computes the transcript hash for the first `ClientHelloInner`, denoted ClientHelloInner1, up to and including the modified HelloRetryRequest. Let transcript_hrr_ech_conf denote the output. Finally, the backend server overwrites the payload of the "encrypted_client_hello" extension with the following string:

~~ hrr_accept_confirmation = HKDF-Expand-Label(HKDF-Extract(0, ClientHelloInner1.random), "hrr ech accept confirmation", transcript_hrr_ech_conf, 8) ~~

In the subsequent ServerHello message, the backend server sends the `accept_confirmation` value as described in Section 7.2.

8. Deployment Considerations

The design of ECH as specified in this document necessarily requires changes to client, client-facing server, and backend server. Coordination between client-facing and backend server requires care, as deployment mistakes can lead to compatibility issues. These are discussed in Section 8.1.

Beyond coordination difficulties, ECH deployments may also induce challenges for use cases of information that ECH protects. In particular, use cases which depend on this unencrypted information may no longer work as desired. This is elaborated upon in Section 8.2.

8.1. 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.

8.1.1. 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 TLS server has a certificate for the public name. If server and advertised keys mismatch, the server will reject ECH and respond with "retry_configs". If the server does not understand the "encrypted_client_hello" extension at all, it will ignore it as required by Section 4.1.2 of [RFC8446]. Provided the server can present a certificate valid for the public name, the client can safely retry with updated settings, as described in Section 6.1.6.

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.

In order to ensure that the retry mechanism works successfully, servers SHOULD ensure that every endpoint which might receive a TLS connection is provisioned with an appropriate certificate for the public name. This is especially important during periods of server reconfiguration when different endpoints might have different configurations.

8.1.2. Middleboxes

The requirements in [RFC8446], Section 9.3 which require proxies to act as conforming TLS client and server provide interoperability with TLS-terminating proxies even in cases where the server supports ECH but the proxy does not, as detailed below.

The proxy must ignore unknown parameters and generate its own `ClientHello` containing only parameters it understands. Thus, when presenting a certificate to the client or sending a `ClientHello` to the server, the proxy will act as if connecting to the `ClientHelloOuter` server_name, which SHOULD match the public name (see Section 6.1) without echoing the "encrypted_client_hello" extension.

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 6.1.6 or result in a connection failure. A proxy which is not authoritative for the public name cannot forge a signal to disable ECH.

8.2. Deployment Impact

Some use cases which depend on information ECH encrypts may break with the deployment of ECH. The extent of breakage depends on a number of external factors, including, for example, whether ECH can be disabled, whether or not the party disabling ECH is trusted to do so, and whether or not client implementations will fall back to TLS without ECH in the event of disablement.

Depending on implementation details and deployment settings, use cases which depend on plaintext TLS information may require fundamentally different approaches to continue working. For example, in managed enterprise settings, one approach may be to disable ECH entirely via group policy and for client implementations to honor this action. Server deployments which depend on SNI -- e.g., for load balancing -- may no longer function properly without updates; the nature of those updates is out of scope of this specification.

In the context of Section 6.1.6, another approach may be to intercept and decrypt client TLS connections. The feasibility of alternative solutions is specific to individual deployments.

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 Section 7.1 of [HPKE])
- KDF: HKDF-SHA256 (see Section 7.2 of [HPKE])
- AEAD: AES-128-GCM (see Section 7.3 of [HPKE])

10. Security Considerations

This section contains security considerations for ECH.

10.1. Security and Privacy Goals

ECH considers two types of attackers: passive and active. Passive attackers can read packets from the network, but 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 also 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 [RFC3552] for TLS 1.3 [RFC8446].

Passive and active attackers can exist anywhere in the network, including between the client and client-facing server, as well as between the client-facing and backend servers when running ECH in split mode. However, for split mode in particular, ECH makes two additional assumptions:

1. The channel between each client-facing and each backend server is authenticated such that the backend server only accepts messages from trusted client-facing servers. The exact mechanism for establishing this authenticated channel is out of scope for this document.
2. The attacker cannot correlate messages between a client and client-facing server with messages between client-facing and backend server. Such correlation could allow an attacker to link information unique to a backend server, such as their server name or IP address, with a client's encrypted `ClientHelloInner`. Correlation could occur through timing analysis of messages across the client-facing server, or via examining the contents of messages sent between client-facing and backend servers. The exact mechanism for preventing this sort of correlation is out of scope for this document.

Given this threat model, the primary goals of ECH are as follows.

1. Security preservation. Use of ECH does not weaken the security properties of TLS without ECH.
2. Handshake privacy. TLS connection establishment to a server name within an anonymity set is indistinguishable from a connection to any other server name within the anonymity set. (The anonymity set is defined in Section 1.)
3. Downgrade resistance. An attacker cannot downgrade a connection that attempts to use ECH to one that does not use ECH.

These properties were formally proven in [ECH-Analysis].

With regards to handshake privacy, client-facing server configuration determines the size of the anonymity set. For example, if a client-facing server uses distinct `ECHConfig` values for each server name, 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 server names as possible. An attacker can distinguish two server names that have different `ECHConfig` values based on the `ECHClientHello.config_id` value.

This also means public information in a TLS handshake should be consistent across server names. For example, if a client-facing server services many backend origin server names, only one of which supports some cipher suite, it may be possible to identify that server name based on the contents of the unencrypted handshake message. Similarly, if a backend origin reuses KeyShare values, then that provides a unique identifier for that server.

Beyond these primary security and privacy goals, ECH also aims to hide, to some extent, the fact that it is being used at all. Specifically, the GREASE ECH extension described in Section 6.2 does not change the security properties of the TLS handshake at all. Its goal is to provide "cover" for the real ECH protocol (Section 6.1), as a means of addressing the "do not stick out" requirements of [RFC8744]. See Section 10.10.4 for details.

10.2. Unauthenticated and Plaintext DNS

ECH supports delivery of configurations through the DNS using SVCB or HTTPS records without requiring any verifiable authenticity or provenance information [RFCYYY1]. 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 configurations (so that the client encrypts data to them) or strip the ECH configurations 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 substitute a unique IP address for each DNS name that was looked up. Thus, using DNS records without additional authentication 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 encrypted DNS transport is also a defense 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.

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 (this may not be possible if clients use the operating system resolver rather than doing their own resolution).

`ECHConfig` rotation rate is also an issue for non-malicious servers, which may want to rotate keys frequently to limit exposure if the key is compromised. Rotating too frequently limits the client anonymity set. In practice, servers which service many server names and thus have high loads are the best candidates to be client-facing servers and so anonymity sets will typically involve many connections even with fairly fast rotation intervals.

10.4. Ignored Configuration Identifiers and Trial Decryption

Ignoring 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 a randomly generated `config_id` in the `ECHClientHello`. 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, ignoring 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 potential damage caused by such attacks.

Unless specified by the application using (D)TLS or externally configured, implementations MUST NOT use this mode.

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 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 that 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 10.10.4.

10.6. Inner ClientHello

Values which depend on the contents of `ClientHelloInner`, such as the true server name, can influence how client-facing servers process this message. In particular, timing side channels can reveal information about the contents of `ClientHelloInner`. Implementations should take such side channels into consideration when reasoning about the privacy properties that ECH provides.

10.7. 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 Online

Certificate Status Protocol (OCSP) or Certificate Revocation List (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.

Attacks that rely on non-ECH traffic to infer server identity in an ECH connection are out of scope for this document. For example, a client that connects to a particular host prior to ECH deployment may later resume a connection to that same host after ECH deployment. An adversary that observes this can deduce that the ECH-enabled connection was made to a host that the client previously connected to and which is within the same anonymity set.

10.8. Cookies

Section 4.2.2 of [RFC8446] defines a cookie value that servers may send in HelloRetryRequest for clients to echo in the second `ClientHello`. While ECH encrypts the cookie in the second `ClientHelloInner`, the backend server's HelloRetryRequest is unencrypted. This means differences in cookies between backend servers, such as lengths or cleartext components, may leak information about the server identity.

Backend servers in an anonymity set SHOULD NOT reveal information in the cookie which identifies the server. This may be done by handling HelloRetryRequest statefully, thus not sending cookies, or by using the same cookie construction for all backend servers.

Note that, if the cookie includes a key name, analogous to Section 4 of [RFC5077], this may leak information if different backend servers issue cookies with different key names at the time of the connection. In particular, if the deployment operates in split mode, the backend servers may not share cookie encryption keys. Backend servers may mitigate this either by handling key rotation with trial decryption or by coordinating to match key names.

10.9. 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). These mechanisms do not interfere because the backend server only signals ECH acceptance in TLS 1.3 or higher.

10.10. Comparison Against Criteria

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

10.10.1. 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`.

10.10.2. 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 of private keys by publishing different DNS records containing `ECHConfig` values with different public keys using a short TTL.

10.10.3. SNI-Based Denial-of-Service Attacks

This design requires servers to decrypt `ClientHello` messages with `ECHClientHello` 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 transport connections an attacker can open.

10.10.4. Do Not Stick Out

As a means of reducing the impact of network ossification, [RFC8744] recommends SNI-protection mechanisms be designed in such a way that network operators do not differentiate connections using the mechanism from connections not using the mechanism. To that end, ECH is designed to resemble a standard TLS handshake as much as possible. The most obvious difference is the extension itself: as long as middleboxes ignore it, as required by [RFC8446], the rest of the handshake is designed to look very much as usual.

The GREASE ECH protocol described in Section 6.2 provides a low-risk way to evaluate the deployability of ECH. It is designed to mimic the real ECH protocol (Section 6.1) without changing the security properties of the handshake. The underlying theory is that if GREASE ECH is deployable without triggering middlebox misbehavior, and real ECH looks enough like GREASE ECH, then ECH should be deployable as well. Thus, the strategy for mitigating network ossification is to deploy GREASE ECH widely enough to disincentivize differential treatment of the real ECH protocol by the network.

Ensuring that networks do not differentiate between real ECH and GREASE ECH may not be feasible for all implementations. While most middleboxes will not treat them differently, some operators may wish to block real ECH usage but allow GREASE ECH. This specification aims to provide a baseline security level that most deployments can achieve easily while providing implementations enough flexibility to achieve stronger security where possible. Minimally, real ECH is designed to be indifferentiable from GREASE ECH for passive adversaries with following capabilities:

1. The attacker does not know the `ECHConfigList` used by the server.
2. The attacker keeps per-connection state only. In particular, it does not track endpoints across connections.

Moreover, real ECH and GREASE ECH are designed so that the following features do not noticeably vary to the attacker, i.e., they are not distinguishers:

1. the code points of extensions negotiated in the clear, and their order;
2. the length of messages; and
3. the values of plaintext alert messages.

This leaves a variety of practical differentiators out-of-scope. including, though not limited to, the following:

1. the value of the configuration identifier;
2. the value of the outer SNI;
3. the TLS version negotiated, which may depend on ECH acceptance;
4. client authentication, which may depend on ECH acceptance; and
5. HRR issuance, which may depend on ECH acceptance.

These can be addressed with more sophisticated implementations, but some mitigations require coordination between the client and server, and even across different client and server implementations. These mitigations are out-of-scope for this specification.

10.10.5. Maintain Forward Secrecy

This design does not provide forward secrecy for the inner `ClientHello` 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 regularly.

10.10.6. 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 allowing the backend origin server to hide behind the client-facing server without the client-facing server decrypting and reencrypting the connection.

Conversely, if the DNS records used for configuration are authenticated, e.g., via DNSSEC, spoofing a client-facing server operating in split mode is not possible. See Section 10.2 for more details regarding plaintext DNS.

Authenticating the `ECHConfig` 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.

10.10.7. 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.

10.11. Padding Policy

Variations in the length of the `ClientHelloInner` ciphertext could leak information about the corresponding plaintext. Section 6.1.3 describes a RECOMMENDED padding mechanism for clients aimed at reducing potential information leakage.

10.12. 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.

10.12.1. 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.

10.12.2. 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 using the same HPKE context for both `ClientHello` messages. The attacker does not possess the context's keys, so it cannot generate a valid encryption of the second inner `ClientHello`.

If the attacker could manipulate the second `ClientHello`, 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.

10.12.3. 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
      + ech_outer_extensions(pre_shared_key)
   + pre_shared_key
               -------->
                     (intercept)
                     ClientHello
                     + key_share
                     + ech
                        + 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. 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`. 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.

10.12.4. ClientHelloInner Packet Amplification Mitigation

Client-facing servers must decompress EncodedClientHelloInners. A malicious attacker may craft a packet which takes excessive resources to decompress or may be much larger than the incoming packet:

- If looking up a `ClientHelloOuter` extension takes time linear in the number of extensions, the overall decoding process would take O(M*N) time, where M is the number of extensions in `ClientHelloOuter` and N is the size of OuterExtensions.
- If the same `ClientHelloOuter` extension can be copied multiple times, an attacker could cause the client-facing server to construct a large `ClientHelloInner` by including a large extension in `ClientHelloOuter` of length L and an OuterExtensions list referencing N

copies of that extension. The client-facing server would then use O(N*L) memory in response to O(N+L) bandwidth from the client. In split mode, an O(N*L)-sized packet would then be transmitted to the backend server.

ECH mitigates this attack by requiring that OuterExtensions be referenced in order, that duplicate references be rejected, and by recommending that client-facing servers use a linear scan to perform decompression. These requirements are detailed in Section 5.1.

11. IANA Considerations

11.1. Update of the TLS ExtensionType Registry

IANA has created the following entries in the existing "TLS ExtensionType Values" registry (defined in [RFC8446]):

1. encrypted_client_hello (0xfe0d), with "TLS 1.3" column values set to "CH, HRR, EE", "DTLS-Only" column set to "N", and "Recommended" column set to "Y".
2. ech_outer_extensions (0xfd00), with the "TLS 1.3" column values set to "CH", "DTLS-Only" column set to "N", "Recommended" column set to "Y", and the "Comment" column set to "Only appears in inner CH."

11.2. Update of the TLS Alert Registry

IANA has created an entry, ech_required (121) in the existing "TLS Alerts" registry (defined in [RFC8446]), with the "DTLS-OK" column set to "Y".

11.3. ECH Configuration Extension Registry

IANA has created a new "TLS ECHConfig Extension" registry in a new "TLS Encrypted Client Hello (ECH) Configuration Extensions" registry group. New registrations will list the following attributes:

Value: The two-byte identifier for the ECHConfigExtension, i.e., the ECHConfigExtensionType

Extension Name: Name of the ECHConfigExtension

Recommended: A "Y" or "N" value indicating if the TLS Working Group recommends that the extension be supported. This column is assigned a value of "N" unless explicitly requested. Adding a value of "Y" requires Standards Action [RFC8126].

Reference: The specification where the ECHConfigExtension is defined

Notes: Any notes associated with the entry

New entries in the "TLS ECHConfig Extension" registry are subject to the Specification Required registration policy ([RFC8126], Section 4.6), with the policies described in [RFC8447], Section 17. IANA has added the following note to the "TLS ECHConfig Extension" registry:

Note: The role of the designated expert is described in RFC 8447. The designated expert [RFC8126] ensures that the specification is publicly available. It is sufficient to have an Internet-Draft (that is posted and never published as an RFC) or a document from another standards body, industry consortium, university site, etc. The expert may provide more in-depth reviews, but their approval should not be taken as an endorsement of the extension.

This document defines several Reserved values for ECH configuration extensions to be used for "greasing" as described in Section 6.2.2.

The initial contents for this registry consists of multiple reserved values with the following attributes, which are repeated for each registration:

Value: 0x0000, 0x1A1A, 0x2A2A, 0x3A3A, 0x4A4A, 0x5A5A, 0x6A6A, 0x7A7A, 0x8A8A, 0x9A9A, 0xAAAA, 0xBABA, 0xCACA, 0xDADA, 0xEAEA, 0xFAFA

Extension Name: RESERVED

Recommended: Y

Reference: RFC 9849

Notes: GREASE entries

12. References

12.1. Normative References

[HPKE] Barnes, R., Bhargavan, K., Lipp, B., and C. Wood, "Hybrid Public Key Encryption", RFC 9180, DOI 10.17487/RFC9180, February 2022, <https://www.rfc-editor.org/info/rfc9180>.

[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>.

[RFC5890] Klensin, J., "Internationalized Domain Names for Applications (IDNA): Definitions and Document Framework", RFC 5890, DOI 10.17487/RFC5890, August 2010, <https://www.rfc-editor.org/info/rfc5890>.

[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>.

[RFC8126] Cotton, M., Leiba, B., and T. Narten, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 8126, DOI 10.17487/RFC8126, June 2017, <https://www.rfc-editor.org/info/rfc8126>.

[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>.

[RFC8447] Salowey, J. and S. Turner, "IANA Registry Updates for TLS and DTLS", RFC 8447, DOI 10.17487/RFC8447, August 2018, <https://www.rfc-editor.org/info/rfc8447>.

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

[RFC9460] Schwartz, B., Bishop, M., and E. Nygren, "Service Binding and Parameter Specification via the DNS (SVCB and HTTPS Resource Records)", RFC 9460, DOI 10.17487/RFC9460, November 2023, <https://www.rfc-editor.org/info/rfc9460>.

[RFC9525] Saint-Andre, P. and R. Salz, "Service Identity in TLS", RFC 9525, DOI 10.17487/RFC9525, November 2023, <https://www.rfc-editor.org/info/rfc9525>.

12.2. Informative References

[DNS-TERMS] Hoffman, P. and K. Fujiwara, "DNS Terminology", BCP 219, RFC 9499, DOI 10.17487/RFC9499, March 2024, <https://www.rfc-editor.org/info/rfc9499>.

[ECH-Analysis] Bhargavan, K., Cheval, V., and C. Wood, "A Symbolic Analysis of Privacy for TLS 1.3 with Encrypted Client Hello", CCS '22: Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, pp. 365-379, DOI 10.1145/3548606.3559360, November 2022, <https://www.cs.ox.ac.uk/people/vincent.cheval/publis/BCW-ccs22.pdf>.

[PROTECTED-SNI] Oku, K., "TLS Extensions for Protecting SNI", Work in Progress, Internet-Draft, draft-kazuho-protected-sni-00, 18 July 2017, <https://datatracker.ietf.org/doc/html/draft-kazuho-protected-sni-00>.

[RFC3552] Rescorla, E. and B. Korver, "Guidelines for Writing RFC Text on Security Considerations", BCP 72, RFC 3552, DOI 10.17487/RFC3552, July 2003, <https://www.rfc-editor.org/info/rfc3552>.

[RFC3986] Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax", STD 66, RFC 3986, DOI 10.17487/RFC3986, January 2005, <https://www.rfc-editor.org/info/rfc3986>.

[RFC5077] Salowey, J., Zhou, H., Eronen, P., and H. Tschofenig, "Transport Layer Security (TLS) Session Resumption without Server-Side State", RFC 5077, DOI 10.17487/RFC5077, January 2008, <https://www.rfc-editor.org/info/rfc5077>.

[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>.

[RFC9250] Huitema, C., Dickinson, S., and A. Mankin, "DNS over Dedicated QUIC Connections", RFC 9250, DOI 10.17487/RFC9250, May 2022, <https://www.rfc-editor.org/info/rfc9250>.

[RFCYYY1] Schwartz, B., Bishop, M., and E. Nygren, "Bootstrapping TLS Encrypted ClientHello with DNS Service Bindings", RFC YYY1, DOI 10.17487/RFCYYY1, December 2025, <https://www.rfc-editor.org/info/rfcYYY1>.

[WHATWG-IPV4] WHATWG, "URL - IPv4 Parser", WHATWG Living Standard, May 2021, <https://url.spec.whatwg.org/#concept-ipv4-parser>.

Appendix A. Linear-Time Outer Extension Processing

The following procedure processes the "ech_outer_extensions" extension (see Section 5.1) in linear time, ensuring that each referenced extension in the `ClientHelloOuter` is included at most once:

1. Let I be initialized to zero and N be set to the number of extensions in `ClientHelloOuter`.
2. For each extension type, E, in OuterExtensions:
   - If E is "encrypted_client_hello", abort the connection with an "illegal_parameter" alert and terminate this procedure.

- While I is less than N and the I-th extension of `ClientHelloOuter` does not have type E, increment I.
- If I is equal to N, abort the connection with an "illegal_parameter" alert and terminate this procedure.
- Otherwise, the I-th extension of `ClientHelloOuter` has type E. Copy it to the `EncodedClientHelloInner` and increment I.

Acknowledgements

This document draws extensively from ideas in [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
Independent
Email: ekr@rtfm.com

Kazuho Oku
Fastly
Email: kazuhooku@gmail.com

Nick Sullivan
Cryptography Consulting LLC
Email: nicholas.sullivan+ietf@gmail.com

Christopher A. Wood
Apple
Email: caw@heapingbits.net