Workgroup: tls
Internet-Draft: draft-ietf-tls-esni-12
Published: 7 July 2021
Intended Status: Standards Track
Expires: 8 January 2022
Authors: E. Rescorla (RTFM, Inc.)
K. Oku (Fastly)
N. Sullivan (Cloudflare)
C.A. Wood (Cloudflare)

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.

Discussion Venues

This note is to be removed before publishing as an RFC.

Source for this draft and an issue tracker can be found at https://github.com/tlswg/draft-ietf-tls-esni.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on 8 January 2022.

Copyright Notice

Table of Contents

1. Introduction

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

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

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

the SNI remains the primary explicit signal used to determine the server's identity.

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

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

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

2. Conventions and Definitions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here. All TLS notation comes from [RFC8446], Section 3.

3. Overview

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

3.1. Topologies

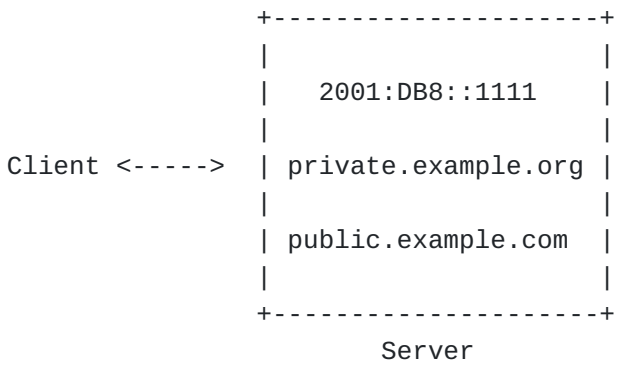

Figure 1: Shared Mode Topology

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.

Figure 2: Split Mode Topology

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

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

3.2. Encrypted ClientHello (ECH)

ECH allows the client to encrypt sensitive ClientHello extensions, e.g., SNI, ALPN, etc., under the public key of the client-facing server. This requires the client-facing server to publish the public key and metadata it uses for ECH for all the domains for which it serves directly or indirectly (via Split Mode). This document defines the format of the ECH encryption public key and metadata,

referred to as an ECH configuration, and delegates DNS publication details to [HTTPS-RR], though other delivery mechanisms are possible. In particular, if some of the clients of a private server are applications rather than Web browsers, those applications might have the public key and metadata preconfigured.

When a client wants to establish a TLS session with the backend server, it constructs its ClientHello as indicated in Section 6.1. We will refer to this as the ClientHelloInner message. The client encrypts this message using the public key of the ECH configuration. It then constructs a new ClientHello, the ClientHelloOuter, with innocuous values for sensitive extensions, e.g., SNI, ALPN, etc., and with an "encrypted_client_hello" extension, which this document defines (Section 5). The extension's payload carries the encrypted ClientHelloInner and specifies the ECH configuration used for encryption. Finally, it sends ClientHelloOuter to the server.

Upon receiving the ClientHelloOuter, a TLS server takes one of the following actions:

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

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

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

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

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 draft-08 of HPKE for public key encryption [I-D.irtf-cfrg-hpke]. The ECH configuration is defined by the following ECHConfig structure.

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

struct {
    HpkeKdfId kdf_id;
    HpkeAeadId aead_id;
} HpkeSymmetricCipherSuite;

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

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

struct {
    uint16 version;
    uint16 length;
    select (ECHConfig.version) {
      case 0xfe0c: ECHConfigContents contents;
    }
} ECHConfig;
```

The structure contains the following fields:

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

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

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

The ECHConfigContents structure contains the following fields:

key_config A HpkeKeyConfig structure carrying the configuration information associated with the HPKE public key. Note that this

structure contains the config_id field, which applies to the entire ECHConfigContents. Sites MUST NOT publish two different ECHConfigContents values with the same HpkeKeyConfig value. The RECOMMENDED technique for choosing config_id is to choose a random byte. This process is repeated if this config_id matches that of any valid ECHConfig, which could include any ECHConfig that has been recently removed from active use.

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 6.1.4.

Clients MUST ignore any ECHConfig structure whose public_name is not parsable as a dot-separated sequence of LDH labels, as defined in [RFC5890], Section 2.3.1 or which begins or end with an ASCII dot.

Clients SHOULD ignore the ECHConfig if it contains an encoded IPv4 address. To determine if a public_name value is an IPv4 address, clients can invoke the IPv4 parser algorithm in [WHATWG-IPV4]. It returns a value when the input is an IPv4 address.

See Section 6.1.4.3 for how the client interprets and validates the public_name.

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

[[OPEN ISSUE: determine if clients should enforce a 63-octet label limit for public_name]] [[OPEN ISSUE: fix reference to WHATWG-IPV4]]

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.

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

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

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

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

```
ECHConfig ECHConfigList<1..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 Extensions

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

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

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

5. The "encrypted_client_hello" Extension

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(0xfe0c), (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. 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 ClientHelloInner structure, encrypted using HPKE as described in Section 6.1.

When the client offers the "encrypted_client_hello" extension, if the payload is the outer variant, then the server MAY include an "encrypted_client_hello" extension in its EncryptedExtensions message 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 in subsequent connection attempts. 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

Some TLS 1.3 extensions can be quite large, thus repeating them in the ClientHelloInner and ClientHelloOuter can lead to an excessive overall size. One pathological example is "key_share" with post-quantum algorithms. To reduce the impact of duplicated extensions, the client may use the "ech_outer_extensions" extension.

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

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

OuterExtensions consists of one or more ExtensionType values, each of which reference an extension in ClientHelloOuter. The extensions in OuterExtensions MUST appear in ClientHelloOuter in the same relative order, however, there is no requirement that they be continguous. For example, OuterExtensions may contain extensions A, B, C, while ClientHelloOuter contains extensions A, D, B, C, E, F.

The "ech_outer_extensions" extension is only used for compressing the ClientHelloInner. It can only be included in EncodedClientHelloInner, and MUST NOT be sent in either ClientHelloOuter or ClientHelloInner.

When sending ClientHello, the client first computes ClientHelloInner, including any PSK binders. It then computes a new value, the EncodedClientHelloInner, which is the following structure:

```
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. Note this field uses the ClientHello structure, defined in Section 4.1.2 of [RFC8446] which does not include the Handshake structure's four byte header. The zeros field MUST be all zeroes.

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

Finally, 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.
- "encrypted_client_hello" appears in OuterExtensions.
- The extensions in ClientHelloOuter corresponding to those in OuterExtensions do not occur in the same order.

Implementations SHOULD bound the time to compute a ClientHelloInner proportionally to the ClientHelloOuter size. If the cost are disproportionately large, a malicious client could exploit this in a denial of service attack. Appendix B 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 reconstructed ClientHelloInner (see Section 10.11.3), 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], which matches the ClientHelloOuter except 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 the four-byte header from the Handshake structure.

The client follows the procedure in Section 6.1.1 to first construct ClientHelloOuterAAD with a placeholder payload field, then replace the field with the encrypted value to compute ClientHelloOuter.

The server then receives ClientHelloOuter and computes ClientHelloOuterAAD by making a copy and replacing the portion corresponding to the payload field with zeros.

The payload and the placeholder strings have the same length, so it is not necessary for either side to recompute length prefixes when applying the above transformations.

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

6. Client Behavior

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 GREASE 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 10.9.4 for an explanation.) The client offers ECH if it is in possession of a compatible ECH configuration and sends GREASE ECH 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.contents.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. Finally, it constructs the ClientHelloOuter message just as it does a standard ClientHello, with the exception of the following rules:

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

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

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 10.11.1.)

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

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 10.11.3.) 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.

8. It MUST include an "encrypted_client_hello" extension with a payload constructed as described in Section 6.1.1.

Note that these rules may change in the presence of an application profile specifying otherwise.

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.

6.1.1. Encrypting the ClientHello

To construct the "encrypted_client_hello", the client first determines the encapsulated key and HPKE encryption context. If constructing the first ClientHelloOuter, it computes them as:

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

If constructing the second ClientHelloOuter (Section 6.1.5), it reuses the encryption context computed for the first ClientHelloOuter, and sets enc to the empty string. Note that 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 10.11.2.

The client then computes ClientHelloOuterAAD (Section 5.2) by constructing a ClientHello with all other extensions determined as in Section 6.1.

Next, 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 fills in an "encrypted_client_hello" extension with the outer variant of ECHClientHello with the following values:

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

If optional configuration identifiers (see Section 10.4) are used, config_id SHOULD be set to a randomly generated byte in the first ClientHelloOuter and MUST be left unchanged for the second ClientHelloOuter.

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

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

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 the server were sending this extension without solicitation, which would violate the extension rules described in [RFC8446]. Sending a GREASE "pre_shared_key" extension in the ClientHelloOuter makes 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

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

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

In contrast, clients do not know the longest SNI value in the client-facing server's anonymity set without server input. Clients SHOULD use the ECHConfig's maximum_name_length field as follows, where L is the maximum_name_length value.

1. If the ClientHelloInner contained a "server_name" extension with a name of length D, add max(0, L - 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 L + 9 bytes of padding. This is the length of a "server_name" extension with an L-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. Handling the Server Response

As described in Section 7, the server MAY either accept ECH and use ClientHelloInner or reject it and use ClientHelloOuter. In handling the server's response, the client's first step is to determine which value was used.

If the server replied with a HelloRetryRequest, then the client proceeds as described in Section 6.1.5. Otherwise, if the server replied with a ServerHello, then the client checks if the last 8 bytes of ServerHello.random are equal to accept_confirmation as defined in Section 7.2. If so, then it presumes acceptance. Otherwise, the client presumes rejection.

6.1.4.1. Accepted ECH

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

6.1.4.2. Rejected ECH

If the server used ClientHelloOuter, the client proceeds with the handshake, authenticating for ECHConfig.contents.public_name as described in Section 6.1.4.3. If authentication or the handshake fails, the client MUST return a failure to the calling application. It MUST NOT use the retry configurations.

Otherwise, if both authentication and the handshake complete successfully, the client MUST abort the connection with an "ech_required" alert. It then processes the "retry_configs" field from the server's "encrypted_client_hello" extension.

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

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

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

If the server negotiates an earlier version of TLS, or if it does not provide an "encrypted_client_hello" extension in EncryptedExtensions, the client proceeds with the handshake, authenticating for ECHConfig.contents.public_name as described in Section 6.1.4.3. If an earlier version was negotiated, the client MUST NOT enable the False Start optimization [RFC7918] for this handshake. If authentication or the handshake fails, the client MUST return a failure to the calling application. It MUST NOT treat this as a secure signal to disable ECH.

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

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

6.1.4.3. Authenticating for the Public Name

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

- The client MUST verify that the certificate is valid for 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. Clients that incorporate DNS names and IP addresses into the same syntax (e.g. [RFC3986], Section 7.4 and [WHATWG-IPV4]) MUST reject names that would be interpreted as IPv4 addresses. Clients that enforce

this by checking and rejecting encoded IPv4 addresses in ECHConfig.contents.public_name do not need to repeat the check at this layer.

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

6.1.5. Handling HelloRetryRequest

When the server sends a HelloRetryRequest, the client checks for the presence of an "encrypted_client_hello" extension. If none is found, then the client presumes rejection and handles the HelloRetryRequest using ClientHelloOuter. (Note that the client-facing server does not send this extension in its HelloRetryRequest. [[NOTE: This may change, depending on the outcome of issue#450.]]) Otherwise it proceeds as follows.

If the extension's payload has a length other than 8, then the client aborts the handshake with an "decode_error" alert. If the payload length is equal to 8, then the client checks if the payload is equal to hrr_accept_confirmation as defined in Section 7.2. If so, then it presumes acceptance and handles handles the HelloRetryRequest using ClientHelloInner. Otherwise, it presumes rejection.

[[OPEN ISSUE: Depending on what we do for issue#450, it may be appropriate to change the client behavior if the HRR payload is missing or malformed.]]

The client encodes the second ClientHelloInner as in Section 5.1, using the second ClientHelloOuter for any referenced extensions. It then encrypts the new EncodedClientHelloInner value as a second message with the previous HPKE context as described in Section 6.1.1.

6.2. GREASE ECH

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.

When sending a second ClientHello in response to a HelloRetryRequest, the client copies the entire "encrypted_client_hello" extension from the first ClientHello.

[[OPEN ISSUE: The above doesn't match HRR handling for either ECH acceptance or rejection. See issue https://github.com/tlswg/draft-ietf-tls-esni/issues/358.]]

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

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

7. Server Behavior

Servers that support ECH play one of two roles, depending on the payload of the "encrypted_client_hello" extension in the initial ClientHello:

- If ECHClientHello.type is outer, then the server acts as a client-facing server and proceeds as described in Section 7.1 to extract a ClientHelloInner, if available.

- If ECHClientHello.type is inner, then the server acts as a backend server and proceeds as described in Section 7.2.

- Otherwise, 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 using (D)TLS or externally configured on both sides, implementations MUST use the first method.

The server then iterates over the candidate ECHConfig values, attempting to decrypt the "encrypted_client_hello" extension:

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.

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. This connection proceeds as usual, except the server MUST include the "encrypted_client_hello" extension in its EncryptedExtensions with the "retry_configs" field set to one or more ECHConfig structures with up-to-date keys. Servers MAY supply multiple ECHConfig values of different versions. This allows a server to support multiple versions at once.

Note that decryption failure could indicate a GREASE ECH extension (see Section 6.2), 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). 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 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.

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.

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. Compatibility Issues

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

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

8.1. Misconfiguration and Deployment Concerns

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

The retry mechanism repairs inconsistencies, provided the server is authoritative for the public name. If server and advertised keys mismatch, the server will respond with ech_retry_requested. If the server does not understand the "encrypted_client_hello" extension at all, it will ignore it as required by 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 6.1.4.

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

8.2. Middleboxes

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

Depending on whether the client is configured to accept the proxy's certificate as authoritative for the public name, this may trigger the retry logic described in Section 6.1.4 or result in a connection failure. A proxy which is not authoritative for the public name cannot forge a signal to disable ECH.

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

9. Compliance Requirements

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

- KEM: DHKEM(X25519, HKDF-SHA256) (see [I-D.irtf-cfrg-hpke], Section 7.1)
- KDF: HKDF-SHA256 (see [I-D.irtf-cfrg-hpke], Section 7.2)
- AEAD: AES-128-GCM (see [I-D.irtf-cfrg-hpke], Section 7.3)

10. Security Considerations

10.1. Security and Privacy Goals

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

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

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

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

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

cipher suite, it may be possible to identify that host based on the contents of unencrypted handshake messages.

Beyond these primary security and privacy goals, ECH also aims to hide, to some extent, 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 10.9.4 for details.

10.2. Unauthenticated and Plaintext DNS

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

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

10.3. Client Tracking

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

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

10.4. Optional Configuration Identifiers and Trial Decryption

Optional configuration identifiers may be useful in scenarios where clients and client-facing servers do not want to reveal information about the client-facing server in the "encrypted_client_hello" extension. In such settings, clients send 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, support for optional configuration identifiers may exacerbate DoS attacks. Specifically, an adversary may send malicious ClientHello messages, i.e., those which will not decrypt with any known ECH key, in order to force wasteful decryption. Servers that support this feature should, for example, implement some form of rate limiting mechanism to limit the damage caused by such attacks.

Unless specified by the application using (D)TLS or externally configured on both sides, 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 a different ALPN lists to different servers or in different application contexts. A client that treats this context as sensitive SHOULD NOT send context-specific values in ClientHelloOuter.

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

Clients MAY include additional extensions in ClientHelloOuter to avoid signaling unusual behavior to passive observers, provided the choice of value and value itself are not sensitive. See Section 10.9.4.

10.6. Related Privacy Leaks

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

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

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.

10.7. 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 by either handling key rotation with trial decryption, or coordinating to match key names.

10.8. 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.

10.9. Comparison Against Criteria

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

10.9.1. 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.

10.9.2. Avoid Widely Shared Secrets

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

10.9.3. Prevent 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 TCP connections an attacker can open.

10.9.4. 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, our 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. 1. The attacker keeps per-connection state only. In particular, it does not track endpoints across connections. 1. ECH and GREASE ECH are designed so that the following features do not vary: the code points of extensions negotiated in the clear; the length of messages; and 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; 1. the value of the outer SNI; 1. use of the "pre_shared_key" extension in the ClientHelloOuter, which is permitted in GREASE ECH but not real ECH; [[TODO: Remove this differentiator if issue #384 is resolved by a spec change.]] 1. the TLS version negotiated, which may depend on ECH acceptance; 1. client authentication, which may depend on ECH acceptance; and 1. 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. These mitigations are out-of-scope for this specification.

10.9.5. Maintain Forward Secrecy

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

10.9.6. Enable Multi-party Security Contexts

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

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

Authenticating the 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.

10.9.7. 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.

10.10. 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.

10.11. 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.

10.11.1. 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.

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.

10.11.2. 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.

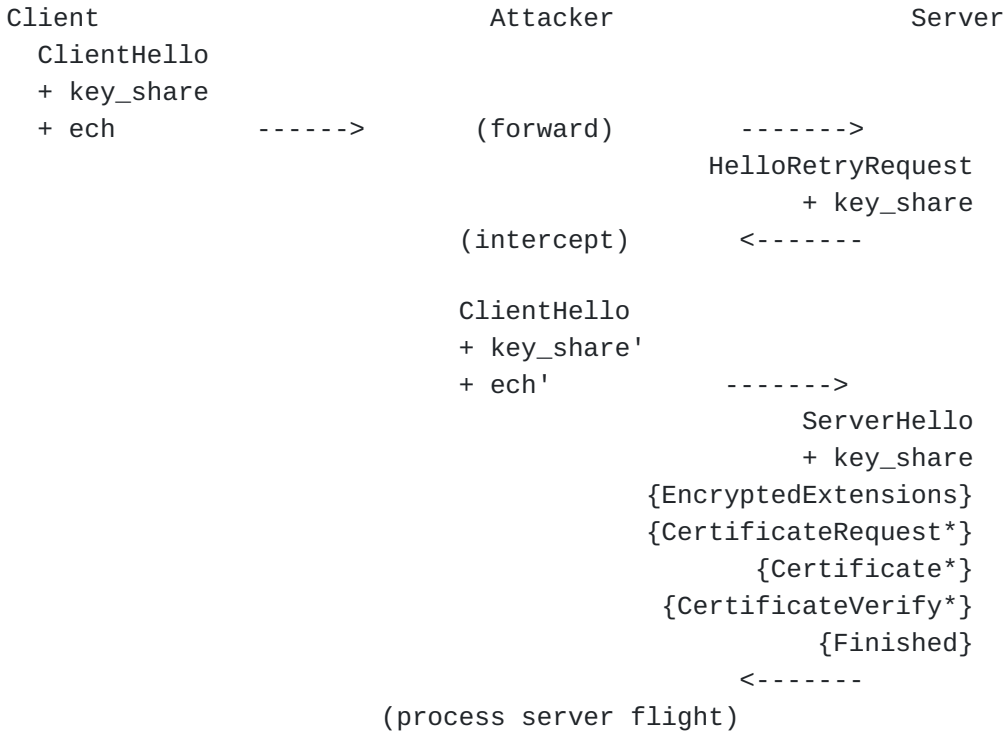

Figure 4: HelloRetryRequest hijack attack

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.

10.11.3. 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.

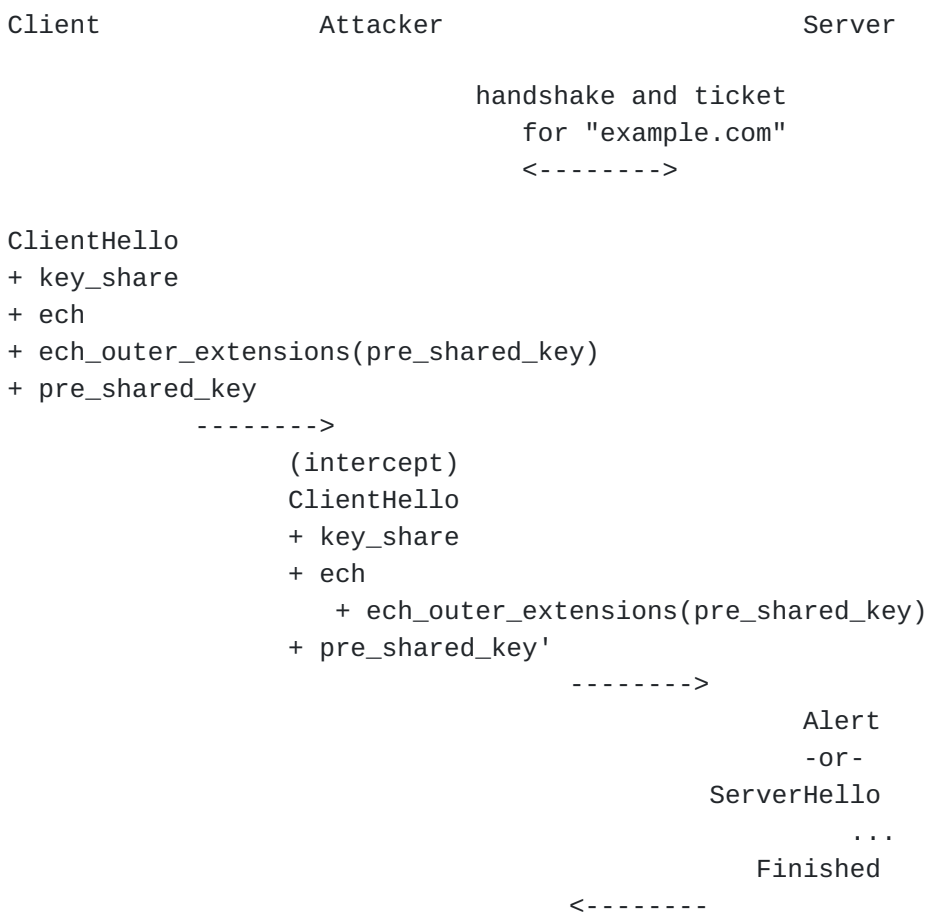

Figure 5: Message flow for malleable ClientHello

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

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

11. IANA Considerations

11.1. Update of the TLS ExtensionType Registry

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

1. encrypted_client_hello(0xfe0c), with "TLS 1.3" column values set to "CH, HRR, EE", and "Recommended" column set to "Yes".

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

11.2. Update of the TLS Alert Registry

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

12. ECHConfig Extension Guidance

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

13. References

13.1. Normative References

[HTTPS-RR] Schwartz, B., Bishop, M., and E. Nygren, "Service binding and parameter specification via the DNS (DNS SVCB and HTTPS RRs)", Work in Progress, Internet-Draft, draft-ietf-dnsop-svcb-https-06, 16 June 2021, <https://www.ietf.org/archive/id/draft-ietf-dnsop-svcb-https-06.txt>.

[I-D.ietf-tls-exported-authenticator] Sullivan, N., "Exported Authenticators in TLS", Work in Progress, Internet-Draft, draft-ietf-tls-exported-authenticator-14, 25 January 2021, <https://www.ietf.org/archive/id/draft-ietf-tls-exported-authenticator-14.txt>.

[I-D.irtf-cfrg-hpke] Barnes, R. L., Bhargavan, K., Lipp, B., and C. A. Wood, "Hybrid Public Key Encryption", Work in Progress, Internet-Draft, draft-irtf-cfrg-hpke-10, 7 July

2021, <https://www.ietf.org/archive/id/draft-irtf-cfrg-hpke-10.txt>.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[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>.

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

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

13.2. Informative References

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

[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>.

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

Appendix A. Alternative SNI Protection Designs

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

A.1. TLS-layer

A.1.1. TLS in Early Data

In this variant, TLS Client Hellos are tunneled within early data payloads belonging to outer TLS connections established with the client-facing server. This requires clients to have established a previous session --- and obtained PSKs --- with the server. The client-facing server decrypts early data payloads to uncover Client Hellos destined for the backend server, and forwards them onwards as

necessary. Afterwards, all records to and from backend servers are forwarded by the client-facing server -- unmodified. This avoids double encryption of TLS records.

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

A.1.2. Combined Tickets

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

A.2. Application-layer

A.2.1. HTTP/2 CERTIFICATE Frames

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

Appendix B. Linear-time Outer Extension Processing

The following procedure processes the "ech_outer_extensions" extension (see Section 5.1) in linear time:

1. Let I be zero and N be 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.

Appendix C. Acknowledgements

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

Appendix D. Change Log

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

Issue and pull request numbers are listed with a leading octothorp.

D.1. Since draft-ietf-tls-esni-11

*Move ClientHello padding to the encoding (#443)

*Align codepoints (#464)

*Relax OuterExtensions checks for alignment with RFC8446 (#467)

*Clarify HRR acceptance and rejection logic (#470)

*Editorial improvements (#468, #465, #462, #461)

D.2. Since draft-ietf-tls-esni-10

*Make HRR confirmation and ECH acceptance explicit (#422, #423)

*Relax computation of the acceptance signal (#420, #449)

*Simplify ClientHelloOuterAAD generation (#438, #442)

*Allow empty enc in ECHClientHello (#444)

*Authenticate ECHClientHello extensions position in ClientHelloOuterAAD (#410)

*Allow clients to send a dummy PSK and early_data in ClientHelloOuter when applicable (#414, #415)

*Compress ECHConfigContents (#409)

*Validate ECHConfig.contents.public_name (#413, #456)

*Validate ClientHelloInner contents (#411)

*Note split-mode challenges for HRR (#418)

*Editorial improvements (#428, #432, #439, #445, #458, #455)

D.3. Since draft-ietf-tls-esni-09

*Finalize HPKE dependency (#390)

*Move from client-computed to server-chosen, one-byte config identifier (#376, #381)

*Rename ECHConfigs to ECHConfigList (#391)

*Clarify some security and privacy properties (#385, #383)

Authors' Addresses

Eric Rescorla
RTFM, Inc.

Email: ekr@rtfm.com

Kazuho Oku
Fastly

Email: kazuhooku@gmail.com

Nick Sullivan
Cloudflare

Email: nick@cloudflare.com

Christopher A. Wood
Cloudflare

Email: caw@heapingbits.net