

tls
Internet-Draft
Intended status: Experimental
Expires: 3 December 2020

E. Rescorla
RTFM, Inc.
K. Oku
Fastly
N. Sullivan
C.A. Wood
Cloudflare
1 June 2020

TLS Encrypted Client Hello
draft-ietf-tls-esni-07

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

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 3 December 2020.

Copyright Notice

Copyright (c) 2020 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<https://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1.	Introduction	3
2.	Conventions and Definitions	4
3.	Overview	4
3.1.	Topologies	4
3.2.	Encrypted ClientHello (ECH)	5
4.	Encrypted ClientHello Configuration	6
5.	The "encrypted_client_hello" extension	7
6.	The "ech_nonce" extension	9
6.1.	Incorporating Outer Extensions	9
7.	Client Behavior	10
7.1.	Sending an encrypted ClientHello	10
7.2.	Recommended Padding Scheme	12
7.3.	Handling the server response	12
7.3.1.	Accepted ECH	13
7.3.2.	Rejected ECH	13
7.3.3.	HelloRetryRequest	14
7.4.	GREASE extensions	15
8.	Client-Facing Server Behavior	16
9.	Compatibility Issues	18
9.1.	Misconfiguration and Deployment Concerns	18
9.2.	Middleboxes	19
10.	Security Considerations	19
10.1.	Why is cleartext DNS OK?	19
10.2.	Client Tracking	20
10.3.	Optional Record Digests and Trial Decryption	20
10.4.	Related Privacy Leaks	20
10.5.	Comparison Against Criteria	21
10.5.1.	Mitigate against replay attacks	21
10.5.2.	Avoid widely-deployed shared secrets	21
10.5.3.	Prevent SNI-based DoS attacks	21
10.5.4.	Do not stick out	21
10.5.5.	Forward secrecy	22
10.5.6.	Proper security context	22
10.5.7.	Split server spoofing	22
10.5.8.	Supporting multiple protocols	22
10.6.	Padding Policy	22
10.7.	Active Attack Mitigations	22
10.7.1.	Client Reaction Attack Mitigation	23
10.7.2.	HelloRetryRequest Hijack Mitigation	24
10.7.3.	Resumption PSK Oracle Mitigation	25
11.	IANA Considerations	25
11.1.	Update of the TLS ExtensionType Registry	26
11.2.	Update of the TLS Alert Registry	26
12.	ECHConfig Extension Guidance	26
13.	References	26
13.1.	Normative References	26

13.2. Informative References	27
Appendix A. Alternative SNI Protection Designs	28
A.1. TLS-layer	28
A.1.1. TLS in Early Data	28
A.1.2. Combined Tickets	29
A.2. Application-layer	29
A.2.1. HTTP/2 CERTIFICATE Frames	29
Appendix B. Total Client Hello Encryption	29
Appendix C. Acknowledgements	30
Authors' Addresses	30

1. Introduction

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

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

The target domain may also be visible through other channels, such as cleartext 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 [I-D.ietf-doh-dns-over-https] 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 WG has extensively studied the problem of protecting the SNI, but has been unable to develop a completely generic solution. [I-D.ietf-tls-sni-encryption] provides a description of the problem space and some of the proposed techniques. One of the more difficult problems is "Do not stick out" ([I-D.ietf-tls-sni-encryption], 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 design in this document takes a different approach: it assumes that private origins will co-locate with or hide behind a provider

(CDN, application server, etc.) which can protect SNI for all of the domains it hosts. As a result, SNI protection does not indicate that the client is attempting to reach a private origin, but only that it is going to a particular service provider, which the observer could already tell from the visible IP address.

The design in this document introduces a new extension, called Encrypted Client Hello (ECH), which allows clients to encrypt the entirety of their ClientHello to a supporting server. This protects the SNI and other potentially sensitive fields, such as the ALPN list. 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 document is designed to operate in one of two primary topologies shown below, which we call "Shared Mode" and "Split Mode"

3.1. Topologies

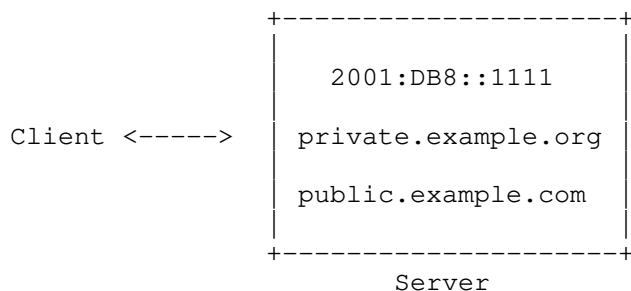


Figure 1: Shared Mode Topology

In Shared Mode, the provider is the origin server for all the domains whose DNS records point to it and clients form a TLS connection directly to that provider, which has access to the plaintext of the connection.

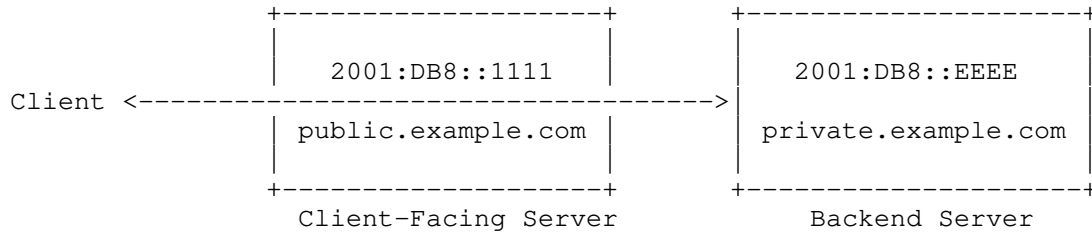


Figure 2: Split Mode Topology

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 backend server, which is the true origin server. The provider does not have access to the plaintext of the connection.

3.2. Encrypted ClientHello (ECH)

ECH works by encrypting the entire ClientHello, including the SNI and any additional extensions such as ALPN. This requires that each provider publish a public key and metadata which is used for ClientHello encryption for all the domains for which it serves directly or indirectly (via Split Mode). This document defines the format of the SNI encryption public key and metadata, referred to as an ECH configuration, and delegates DNS publication details to [HTTPSSVC], 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 form a TLS connection to any of the domains served by an ECH-supporting provider, it constructs a ClientHello in the regular fashion containing the true SNI value (ClientHelloInner) and then encrypts it using the public key for the provider. It then constructs a new ClientHello (ClientHelloOuter) with an innocuous SNI (and potentially innocuous versions of other extensions such as ALPN [RFC7301]) and containing the encrypted ClientHelloInner as an extension. It sends ClientHelloOuter to the server.

Upon receiving ClientHelloOuter, the server can then decrypt ClientHelloInner and either terminate the connection (in Shared Mode) or forward it to the backend server (in Split Mode).

Note that both ClientHelloInner and ClientHelloOuter are both valid, complete ClientHello messages. ClientHelloOuter carries an encrypted representation of ClientHelloInner in a "encrypted_client_hello" extension, defined in Section 5.

4. Encrypted ClientHello Configuration

ClientHello encryption configuration information is conveyed with the following ECHConfigs structure.

```

opaque HpkePublicKey<1..2^16-1>;
uint16 HkpeKemId; // Defined in I-D.irtf-cfrg-hpke
uint16 HkpeKdfId; // Defined in I-D.irtf-cfrg-hpke
uint16 HkpeAeadId; // Defined in I-D.irtf-cfrg-hpke

struct {
    HkpeKdfId kdf_id;
    HkpeAeadId aead_id;
} HpkeCipherSuite;

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

    HpkePublicKey public_key;
    HkpeKemId kem_id;
    HpkeCipherSuite cipher_suites<4..2^16-2>;

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

struct {
    uint16 version;
    uint16 length;
    select (ECHConfig.version) {
        case 0xff07: ECHConfigContents;
    }
} ECHConfig;

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

```

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

The ECHConfig structure contains the following fields:

version The version of the structure. For this specification, that value SHALL be 0xff07. Clients MUST ignore any ECHConfig structure with a version they do not understand.

contents An opaque byte string whose contents depend on the version

of the structure. For this specification, the contents are an ECHConfigContents structure.

The ECHConfigContents structure contains the following fields:

`public_name` The non-empty name of the entity trusted to update these encryption keys. This is used to repair misconfigurations, as described in Section 7.3.

`public_key` The HPKE [I-D.irtf-cfrg-hpke] public key which can be used by the client to encrypt the ClientHello.

`kem_id` The HPKE [I-D.irtf-cfrg-hpke] KEM identifier corresponding to `public_key`. Clients MUST ignore any ECHConfig structure with a key using a KEM they do not support.

`cipher_suites` The list of HPKE [I-D.irtf-cfrg-hpke] AEAD and KDF identifier pairs clients can use for encrypting the ClientHello.

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

`extensions` A list of extensions that the client can take into consideration when generating a ClientHello message. The purpose of the field is to provide room for additional functionality in the future. See Section 12 for guidance on what type of extensions are appropriate for this structure.

The format is 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 which 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 reject the ECHConfig value.

5. The "encrypted_client_hello" extension

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

```
enum {
    encrypted_client_hello(0xff02), (65535)
} ExtensionType;
```

For clients (in ClientHello), this extension contains the following ClientEncryptedCH structure:

```
struct {
    HpkeCipherSuite suite;
    opaque record_digest<0..2^16-1>;
    opaque enc<1..2^16-1>;
    opaque encrypted_ch<1..2^16-1>;
} ClientEncryptedCH;
```

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

record_digest A cryptographic hash of the ECHConfig structure from which the ECH key was obtained, i.e., from the first byte of "version" to the end of the structure. This hash is computed using the hash function associated with "suite", i.e., the corresponding HPKE KDF algorithm hash.

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

encrypted_ch The serialized and encrypted ClientHelloInner structure, AEAD-encrypted using HPKE with the selected KEM, KDF, and AEAD algorithm and key generated as described below.

If the server accepts ECH, it does not send this extension. If it rejects ECH, then it sends the following structure in EncryptedExtensions:

```
struct {
    ECHConfigs retry_configs;
} ServerEncryptedCH;
```

retry_configs An ECHConfigs structure containing one or more ECHConfig structures in decreasing order of preference that the client should use on subsequent connections to encrypt the ClientHelloInner structure.

This protocol also defines the "ech_required" alert, which is sent by the client when it offered an "encrypted_client_hello" extension which was not accepted by the server.

6. The "ech_nonce" extension

When using ECH, the client MUST also add an extension of type "ech_nonce" to the ClientHelloInner (but not to the outer ClientHello). This nonce ensures that the server's encrypted Certificate can only be read by the entity which sent this ClientHello. This extension is defined as follows:

```
enum {
    ech_nonce(0xff03), (65535)
} ExtensionType;

struct {
    uint8 nonce[16];
} ECHNonce;
```

nonce A 16-byte nonce exported from the HPKE encryption context. See Section 7.1 for details about its computation.

Finally, requirements in Section 7 and Section 8 require implementations to track, alongside each PSK established by a previous connection, whether the connection negotiated this extension with the "ech_accept" response type. If so, this is referred to as an "ECH PSK". Otherwise, it is a "non-ECH PSK". This may be implemented by adding a new field to client and server session states.

6.1. Incorporating Outer Extensions

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

```
enum {
    outer_extension(0xff04), (65535)
} ExtensionType;

struct {
    ExtensionType outer_extensions<2..254>;
    uint8 hash<32..255>;
} OuterExtensions;
```

OuterExtensions MUST only be used in ClientHelloInner. It consists of one or more ExtensionType values, each of which reference an extension in ClientHelloOuter, and a digest of the complete ClientHelloInner.

When sending ClientHello, the client first computes ClientHelloInner, including any PSK binders, and then MAY substitute extensions which it knows will be duplicated in ClientHelloOuter. To do so, the client computes a hash H of the entire ClientHelloInner message with the same hash as for the KDF used to encrypt ClientHelloInner. Then, the client removes and replaces extensions from ClientHelloInner with a single "outer_extensions" extension. The list of outer_extensions include those which were removed from ClientHelloInner, in the order in which they were removed. The hash contains the full ClientHelloInner hash H computed above.

This process is reversed by client-facing servers upon receipt. Specifically, the server replaces the "outer_extensions" with extensions contained in ClientHelloOuter. The server then computes a hash H' of the reconstructed ClientHelloInner. If H' does not equal OuterExtensions.hash, the server aborts the connection with an "illegal_parameter" alert.

Clients SHOULD only use this mechanism for extensions which are large. All other extensions SHOULD appear in both ClientHelloInner and ClientHelloOuter even if they have identical values.

7. Client Behavior

7.1. Sending an encrypted ClientHello

In order to send an encrypted ClientHello, the client first determines if it supports the server's chosen KEM, as identified by ECHConfig.kem_id. If one is supported, the client MUST select an appropriate HpkeCipherSuite from the list of suites offered by the server. If the client does not support the corresponding KEM or is unable to select an appropriate group or HpkeCipherSuite, it SHOULD ignore that ECHConfig value and MAY attempt to use another value provided by the server. The client MUST NOT send ECH using HPKE algorithms not advertised by the server.

Given a compatible ECHConfig with fields public_key and kem_id, carrying the HpkePublicKey and KEM identifier corresponding to the server, clients compute an HPKE encryption context as follows:

```
pkR = HPKE.KEM.Unmarshal(ECHConfig.public_key)
enc, context = SetupBaseS(pkR, "tls13-ech")
ech_nonce_value = context.Export("tls13-ech-nonce", 16)
ech_hrr_key = context.Export("tls13-ech-hrr-key", 16)
```

Note that the HPKE algorithm identifiers are those which match the client's chosen preference from ECHConfig.cipher_suites. The client MAY replace any large, duplicated extensions in ClientHelloInner with

the corresponding "outer_extensions" extension, as described in Section 6.1.

The client then generates a ClientHelloInner value. In addition to the normal values, ClientHelloInner MUST also contain:

- * an "ech_nonce" extension, containing "ech_nonce_value" derived above
- * TLS padding [RFC7685] (see Section 7.2)

When offering an encrypted ClientHello, the client MUST NOT offer to resume any non-ECH PSKs. It additionally MUST NOT offer to resume any sessions for TLS 1.2 or below.

The encrypted ClientHello value is then computed as:

```
encrypted_ch = context.Seal("", ClientHelloInner)
```

Finally, the client MUST generate a ClientHelloOuter message containing the "encrypted_client_hello" extension with the values as indicated above. In particular,

- * suite contains the client's chosen HpkeCipherSuite;
- * record_digest contains the digest of the corresponding ECHConfig structure;
- * enc contains the encapsulated key as output by SetupBaseS; and
- * encrypted_ch contains the HPKE encapsulated key (enc) and the ClientHelloInner ciphertext (encrypted_ch_inner).

The client MUST place the value of ECHConfig.public_name in the ClientHelloOuter "server_name" extension. The ClientHelloOuter MUST NOT contain a "cached_info" extension [RFC7924] with a CachedObject entry whose CachedInformationType is "cert", since this indication would divulge the true server name. The remaining contents of the ClientHelloOuter MAY be identical to those in ClientHelloInner but MAY also differ.

7.2. Recommended Padding Scheme

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

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

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

1. If $L > D$, add $L - D$ bytes of padding. This rounds to the server's advertised hint, i.e., ECHConfig.maximum_name_length.
2. Otherwise, add $32 - (D \% 32)$ bytes of padding. This rounds D up to the nearest multiple of 32 bytes.

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.

7.3. Handling the server response

As described in Section 8, the server MAY either accept ECH and use ClientHelloInner or reject it and use ClientHelloOuter. However, there is no indication in ServerHello of which one the server has done and the client must therefore use trial decryption in order to determine this.

7.3.1. Accepted ECH

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

7.3.2. Rejected ECH

If the server used ClientHelloOuter, the client proceeds with the handshake, authenticating for ECHConfig.public_name as described in Section 7.3.2.1. If authentication or the handshake fails, the client MUST return a failure to the calling application. It MUST NOT use the retry keys.

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

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

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

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

If the server negotiates an earlier version of TLS, or if it does not provide an "encrypted_client_hello" extension in EncryptedExtensions, the client proceeds with the handshake, authenticating for ECHConfigContents.public_name as described in Section 7.3.2.1. If an earlier version was negotiated, the client MUST NOT enable the False Start optimization [RFC7918] for this handshake. If authentication or the handshake fails, the client MUST return a failure to the calling application. It MUST NOT treat this as a secure signal to disable ECH.

Otherwise, when the handshake completes successfully with the public name authenticated, the client MUST abort the connection with an

"ech_required" alert. The client can then regard ECH as securely disabled by the server. It SHOULD retry the handshake with a new transport connection and ECH disabled.

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

7.3.2.1. Authenticating for the public name

When the server cannot decrypt or does not process the "encrypted_client_hello" extension, it continues with the handshake using the cleartext "server_name" extension instead (see Section 8). Clients that offer ECH then authenticate the connection with the public name, as follows:

- * If the server resumed a session or negotiated a session that did not use a certificate for authentication, the client MUST abort the connection with an "illegal_parameter" alert. This case is invalid because Section 7.1 requires the client to only offer ECH-established sessions, and Section 8 requires the server to decline ECH-established sessions if it did not accept ECH.
- * The client MUST verify that the certificate is valid for ECHConfigContents.public_name. If invalid, it MUST abort the connection with the appropriate alert.
- * If the server requests a client certificate, the client MUST respond with an empty Certificate message, denoting no client certificate.

Note that authenticating a connection for the public name does not authenticate it for the origin. The TLS implementation MUST NOT report such connections as successful to the application. It additionally MUST ignore all session tickets and session IDs presented by the server. These connections are only used to trigger retries, as described in Section 7.3. This may be implemented, for instance, by reporting a failed connection with a dedicated error code.

7.3.3. HelloRetryRequest

If the server sends a HelloRetryRequest in response to the ClientHello, the client sends a second updated ClientHello per the rules in [RFC8446]. However, at this point, the client does not know whether the server processed ClientHelloOuter or ClientHelloInner, and MUST regenerate both values to be acceptable. Note: if the inner

and outer ClientHellos use different groups for their key shares or differ in some other way, then the HelloRetryRequest may actually be invalid for one or the other ClientHello, in which case a fresh ClientHello MUST be generated, ignoring the instructions in HelloRetryRequest. Otherwise, the usual rules for HelloRetryRequest processing apply.

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

```
pkR = HPKE.KEM.Unmarshal(ECHConfig.public_key)
enc, context = SetupPSKS(pkR, "tls13-ech-hrr", ech_hrr_key, "")
ech_nonce_value = context.Export("tls13-ech-hrr-nonce", 16)
```

Clients then encrypt the second ClientHelloInner using this new HPKE context. In doing so, the encrypted value is also authenticated by ech_hrr_key. The rationale for this is described in Section 10.7.2.

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

```
context = SetupPSKR(ClientEncryptedCH.enc, skR, "tls13-ech-hrr", ech_hrr_key,
"")
ClientHelloInner = context.Open("", ClientEncryptedCH.encrypted_ch)
ech_nonce_value = context.Export("tls13-ech-hrr-nonce", 16)
```

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

7.4. GREASE extensions

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

- * Set the "suite" field to a supported HpkeCipherSuite. 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 "record_digest" field to a randomly-generated string of hash_length bytes, where hash_length is the length of the hash function associated with the chosen HpkeCipherSuite.

- * Set the "enc" field to a randomly-generated valid encapsulated public key output by the HPKE KEM.
- * Set the "encrypted_ch" field to a randomly-generated string of L bytes, where L is the size of the ClientHelloInner message the client would use given an ECHConfig structure, padded according to Section 7.2.

If the server sends an "encrypted_client_hello" extension, the client MUST check the extension syntactically and abort the connection with a "decode_error" alert if it is invalid.

Offering a GREASE extension is not considered offering an encrypted ClientHello for purposes of requirements in Section 7. In particular, the client MAY offer to resume sessions established without ECH.

8. Client-Facing Server Behavior

Upon receiving an "encrypted_client_hello" extension, the client-facing server MUST check that it is able to negotiate TLS 1.3 or greater. If not, it MUST abort the connection with a "handshake_failure" alert.

The ClientEncryptedCH value is said to match a known ECHConfig if there exists an ECHConfig that can be used to successfully decrypt ClientEncryptedCH.encrypted_ch. This matching procedure should be done using one of the following two checks:

1. Compare ClientEncryptedCH.record_digest against cryptographic hashes of known ECHConfig and choose the one that matches.
2. Use trial decryption of ClientEncryptedCH.encrypted_ch with known ECHConfig and choose the one that succeeds.

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

If the ClientEncryptedCH value does not match any known ECHConfig structure, it MUST ignore the extension and proceed with the connection, with the following added behavior:

- * It MUST include the "encrypted_client_hello" extension with the "retry_keys" 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.

- * The server MUST ignore all PSK identities in the ClientHello which correspond to ECH PSKs. ECH PSKs offered by the client are associated with the ECH name. The server was unable to decrypt then ECH name, so it should not resume them when using the cleartext SNI name. This restriction allows a client to reject resumptions in Section 7.3.2.1.

Note that an unrecognized ClientEncryptedCH.record_digest value may be a GREASE ECH extension (see Section 7.4), so it is necessary for servers to proceed with the connection and rely on the client to abort if ECH was required. In particular, the unrecognized value alone does not indicate a misconfigured ECH advertisement (Section 9.1). Instead, servers can measure occurrences of the "ech_required" alert to detect this case.

If the ClientEncryptedCH value matches a known ECHConfig, the server then decrypts ClientEncryptedCH.encrypted_ch, using the private key skR corresponding to ECHConfig, as follows:

```
context = SetupBaseR(ClientEncryptedCH.enc, skR, "tls13-ech")
ClientHelloInner = context.Open("", ClientEncryptedCH.encrypted_ch)
ech_nonce_value = context.Export("tls13-ech-nonce", 16)
ech_hrr_key = context.Export("tls13-ech-hrr-key", 16)
```

If decryption fails, the server MUST abort the connection with a "decrypt_error" alert. Moreover, if there is no "ech_nonce" extension, or if its value does not match the derived ech_nonce, the server MUST abort the connection with a "decrypt_error" alert. Next, the server MUST scan ClientHelloInner for any "outer_extension" extensions and substitute their values with the values in ClientHelloOuter. It MUST first verify that the hash found in the extension matches the hash of the extension to be interpolated in and if it does not, abort the connections with a "decrypt_error" alert.

Upon determining the true SNI, the client-facing server then either serves the connection directly (if in Shared Mode), in which case it executes the steps in the following section, or forwards the TLS connection to the backend server (if in Split Mode). In the latter case, it does not make any changes to the TLS messages, but just blindly forwards them.

If the server sends a NewSessionTicket message, the corresponding ECH PSK MUST be ignored by all other servers in the deployment when not

negotiating ECH, including servers which do not implement this specification.

9. 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 cleartext 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.

9.1. Misconfiguration and Deployment Concerns

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

The retry mechanism repairs inconsistencies, provided the server is authoritative for the public name. If server and advertised keys mismatch, the server will respond with `ech_retry_requested`. If the server does not understand the "encrypted_client_hello" extension at all, it will ignore it as required by [RFC8446]; Section 4.1.2. Provided the server can present a certificate valid for the public name, the client can safely retry with updated settings, as described in Section 7.3.

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

Client-facing servers with non-uniform cryptographic configurations across backend origin servers segment the ECH anonymity set based on these configurations. For example, if a client-facing server hosts *k* backend origin servers, and exactly one of those backend origin servers supports a different set of cryptographic algorithms than the

other $(k - 1)$ servers, it may be possible to identify this single server based on the contents of the ServerHello as this message is not encrypted.

9.2. 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 7.3 or result in a connection failure. A proxy which is not authoritative for the public name cannot forge a signal to disable ECH.

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

10. Security Considerations

10.1. Why is cleartext DNS OK?

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.

10.2. 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 HTTPSSVC RRs with high TTLs, challenging. Clients can help mitigate this problem by flushing any DNS or ECHConfig state upon changing networks.

10.3. Optional Record Digests and Trial Decryption

Optional record digests 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, servers must perform trial decrypt upon receipt of an empty record digest, which may exacerbate DoS attacks. Specifically, an adversary may send malicious ClientHello messages, i.e., those which will not decrypt with any known ECH key, in order to force wasteful decryption. Servers that support this feature should, for example, implement some form of rate limiting mechanism to limit the damage caused by such attacks.

10.4. Related Privacy Leaks

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

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

SHOULD take steps to minimize or protect such requests during certificate validation.

10.5. Comparison Against Criteria

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

10.5.1. Mitigate against replay attacks

Since servers process either ClientHelloInner or ClientHelloOuter, and ClientHelloInner contains an HPKE-derived nonce, it is not possible for an attacker to "cut and paste" the ECH value in a different Client Hello and learn information from ClientHelloInner. This is because the attacker lacks access to the HPKE-derived nonce used to derive the handshake secrets.

10.5.2. Avoid widely-deployed 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.

10.5.3. Prevent SNI-based DoS attacks

This design requires servers to decrypt ClientHello messages with ClientEncryptedCH 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.

10.5.4. Do not stick out

The only explicit signal indicating possible use of ECH is the ClientHello "encrypted_client_hello" extension. Server handshake messages do not contain any signal indicating use or negotiation of ECH. Clients MAY GREASE the "encrypted_client_hello" extension, as described in Section 7.4, which helps ensure the ecosystem handles ECH correctly. Moreover, as more clients enable ECH support, e.g., as normal part of Web browser functionality, with keys supplied by shared hosting providers, the presence of ECH extensions becomes less unusual and part of typical client behavior. In other words, if all

Web browsers start using ECH, the presence of this value will not signal unusual behavior to passive eavesdroppers.

10.5.5. 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.

10.5.6. Proper security context

This design permits servers operating in Split Mode to forward connections directly to backend origin servers, thereby avoiding unnecessary MiTM attacks.

10.5.7. Split server spoofing

Assuming ECH records retrieved from DNS are authenticated, e.g., via DNSSEC or fetched from a trusted Recursive Resolver, spoofing a server operating in Split Mode is not possible. See Section 10.1 for more details regarding cleartext DNS.

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

10.5.8. Supporting 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 protocols.

10.6. Padding Policy

Variations in the length of the ClientHelloInner ciphertext could leak information about the corresponding plaintext. Section 7.2 describes a RECOMMENDED padding mechanism for clients aimed at reducing potential information leakage.

10.7. 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.

10.7.1. 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

The "ech_nonce" extension in the inner ClientHello 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.

10.7.2. 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.

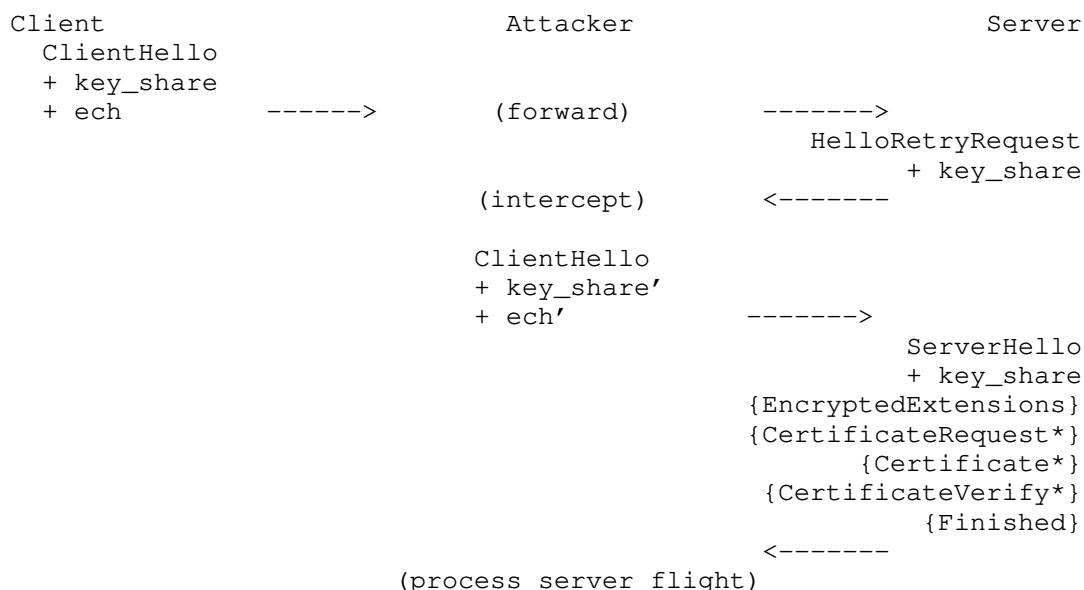


Figure 4: HelloRetryRequest hijack attack

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

If the second ClientHello were not bound to the first, it might be possible for the server to act as an oracle if it required parameters from the first ClientHello to match that of the second ClientHello.

For example, imagine the client's original SNI value in the inner ClientHello is "example.com", and the attacker's hijacked SNI value in its inner ClientHello is "test.com". A server which checks these for equality and changes behavior based on the result can be used as an oracle to learn the client's SNI.

10.7.3. Resumption PSK Oracle Mitigation

This attack uses resumption PSKs as an oracle for dictionary attacks against a given ClientHello's true SNI. To begin, the attacker first interacts with a server to obtain a resumption ticket for a given test domain, such as "test.com". Later, upon receipt of a legitimate ClientHello without a PSK binder, it computes a PSK binder using its own ticket and forwards the resulting ClientHello. Assume the server then validates the PSK binder on the outer ClientHello and chooses connection parameters based on the inner ClientHello. A server which then validates information in the outer ClientHello ticket against information in the inner ClientHello, such as the SNI, introduces an oracle that can be used to test the encrypted SNI value of specific ClientHello messages.

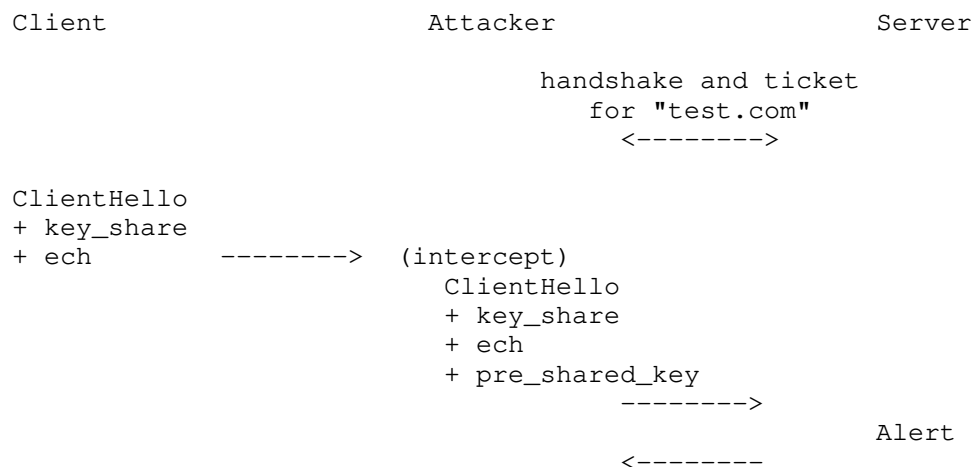


Figure 5: Message flow for resumption and PSK

ECH mitigates against this attack by requiring servers not mix-and-match information from the inner and outer ClientHello. For example, if the server accepts the inner ClientHello, it does not validate binders in the outer ClientHello. This means that ECH PSKs are used within the HPKE encryption envelope.

11. IANA Considerations

11.1. Update of the TLS ExtensionType Registry

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

1. encrypted_client_hello(0xff02), with "TLS 1.3" column values being set to "CH, EE", and "Recommended" column being set to "Yes".
2. ech_nonce(0xff03), with the "TLS 1.3" column values being set to "CH", and "Recommended" column being set to "Yes".
3. outer_extension(0xff04), with the "TLS 1.3" column values being set to "CH", and "Recommended" column being set to "Yes".

11.2. Update of the TLS Alert Registry

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

12. ECHConfig Extension Guidance

Any future information or hints that influence the outer ClientHello SHOULD be specified as ECHConfig extensions, or in an entirely new version of ECHConfig. 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 8). In contrast, the inner ClientHello is the true ClientHello used upon ECH negotiation.

13. References

13.1. Normative References

[HTTPSSVC] Schwartz, B., Bishop, M., and E. Nygren, "Service binding and parameter specification via the DNS (DNS SVCB and HTTPSSVC)", Work in Progress, Internet-Draft, draft-nygren-dnsop-svcb-httpssvc-00, 23 September 2019, <<http://www.ietf.org/internet-drafts/draft-nygren-dnsop-svcb-httpssvc-00.txt>>.

[I-D.ietf-tls-exported-authenticator] Sullivan, N., "Exported Authenticators in TLS", Work in Progress, Internet-Draft, draft-ietf-tls-exported-authenticator-12, 15 May 2020, <<http://www.ietf.org/internet-drafts/draft-ietf-tls-exported-authenticator-12.txt>>.

- [I-D.irtf-cfrg-hpke]
Barnes, R., Bhargavan, K., and C. Wood, "Hybrid Public Key Encryption", Work in Progress, Internet-Draft, draft-irtf-cfrg-hpke-04, 8 May 2020, <<http://www.ietf.org/internet-drafts/draft-irtf-cfrg-hpke-04.txt>>.
- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <<https://www.rfc-editor.org/info/rfc2119>>.
- [RFC7685] Langley, A., "A Transport Layer Security (TLS) ClientHello Padding Extension", RFC 7685, DOI 10.17487/RFC7685, October 2015, <<https://www.rfc-editor.org/info/rfc7685>>.
- [RFC7918] Langley, A., Modadugu, N., and B. Moeller, "Transport Layer Security (TLS) False Start", RFC 7918, DOI 10.17487/RFC7918, August 2016, <<https://www.rfc-editor.org/info/rfc7918>>.
- [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>>.
- [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.ietf-doh-dns-over-https]
Hoffman, P. and P. McManus, "DNS Queries over HTTPS (DoH)", Work in Progress, Internet-Draft, draft-ietf-doh-dns-over-https-14, 16 August 2018, <<http://www.ietf.org/internet-drafts/draft-ietf-doh-dns-over-https-14.txt>>.
- [I-D.ietf-tls-sni-encryption]
Huitema, C. and E. Rescorla, "Issues and Requirements for SNI Encryption in TLS", Work in Progress, Internet-Draft, draft-ietf-tls-sni-encryption-09, 28 October 2019, <<http://www.ietf.org/internet-drafts/draft-ietf-tls-sni-encryption-09.txt>>.

- [I-D.kazuho-protected-sni] Oku, K., "TLS Extensions for Protecting SNI", Work in Progress, Internet-Draft, draft-kazuho-protected-sni-00, 18 July 2017, <<http://www.ietf.org/internet-drafts/draft-kazuho-protected-sni-00.txt>>.
- [RFC7301] Friedl, S., Popov, A., Langley, A., and E. Stephan, "Transport Layer Security (TLS) Application-Layer Protocol Negotiation Extension", RFC 7301, DOI 10.17487/RFC7301, July 2014, <<https://www.rfc-editor.org/info/rfc7301>>.
- [RFC7858] Hu, Z., Zhu, L., Heidemann, J., Mankin, A., Wessels, D., and P. Hoffman, "Specification for DNS over Transport Layer Security (TLS)", RFC 7858, DOI 10.17487/RFC7858, May 2016, <<https://www.rfc-editor.org/info/rfc7858>>.
- [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>>.
- [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>>.

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. Total Client Hello Encryption

The design described here only provides encryption for the SNI, but not for other extensions, such as ALPN. Another potential design would be to encrypt all of the extensions using the same basic structure as we use here for ECH. That design has the following advantages:

- * It protects all the extensions from ordinary eavesdroppers

- * If the encrypted block has its own KeyShare, it does not necessarily require the client to use a single KeyShare, because the client's share is bound to the SNI by the AEAD (analysis needed).

It also has the following disadvantages:

- * The client-facing server can still see the other extensions. By contrast we could introduce another EncryptedExtensions block that was encrypted to the backend server and not the client-facing server.
- * It requires a mechanism for the client-facing server to provide the extension-encryption key to the backend server and thus cannot be used with an unmodified backend server.
- * A conforming middlebox will strip every extension, which might result in a ClientHello which is just unacceptable to the server (more analysis needed).

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.

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