tls
Internet-Draft
Intended status: Experimental
Expires: September 12, 2019

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

March 11, 2019

Encrypted Server Name Indication for TLS 1.3
draft-ietf-tls-esni-03

Abstract

This document defines a simple mechanism for encrypting the Server Name Indication for TLS 1.3.

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

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

Although TLS 1.3 [RFC8446] encrypts most of the handshake, including the server certificate, there are several other channels that allow an on-path attacker to determine the domain name the client is trying to connect to, including:

o Cleartext client DNS queries.

o Visible server IP addresses, assuming the the server is not doing domain-based virtual hosting.

o Cleartext Server Name Indication (SNI) [RFC6066] in ClientHello messages.

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, SNI is an explicit signal used to determine the server's identity. Indirect mechanisms such as traffic analysis also exist.

The TLS WG has extensively studied the problem of protecting 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/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 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, app server, etc.) which is able to activate encrypted SNI (ESNI) for all of the domains it hosts. Thus, the use of encrypted SNI 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 IP address.

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.

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

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

Figure 1: Shared Mode Topology

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

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

Figure 2: Split Mode Topology

In Split Mode, the provider is _not_ the origin server for private domains. Rather the DNS records for private domains point to the provider, but the provider's server just 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. In principle, the provider might not be the origin for any domains, but as a practical matter, it is probably the origin for a large set of innocuous domains, but is also providing protection for some private domains. Note that the backend server can be an unmodified TLS 1.3 server.

3.2. SNI Encryption

First, the provider publishes a public key and metadata which is used for SNI encryption for all the domains for which it serves directly or indirectly (via Split Mode). This document defines a publication mechanism using DNS, but other mechanisms are also 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 ESNI-supporting provider, it sends an "encrypted_server_name" extension, which contains the true extension encrypted under the provider's public key. The provider can then decrypt the extension and either terminate the connection (in Shared Mode) or forward it to the backend server (in Split Mode).

4. Publishing the SNI Encryption Key in the DNS

Publishing ESNI keys in the DNS requires care to ensure correct behavior. There are deployment environments in which a domain is served by multiple server operators who do not manage the ESNI Keys. Because ESNIKeys and A/AAAA lookup are independent, it is therefore possible to obtain an ESNIKeys record which does not match the A/AAAA records. (That is, the host to which an A or AAAA record refers is

not in possession of the ESNI keys.) The design of the system must therefore allow clients to detect and recover from this situation.

Servers operating in Split Mode SHOULD have DNS configured to return the same A (or AAAA) record for all ESNI-enabled servers they service. This yields an anonymity set of cardinality equal to the number of ESNI-enabled server domains supported by a given client-facing server. Thus, even with SNI encryption, an attacker which can enumerate the set of ESNI-enabled domains supported by a client-facing server can guess the correct SNI with probability at least 1/K, where K is the size of this ESNI-enabled server anonymity set. This probability may be increased via traffic analysis or other mechanisms.

The following sections describe a DNS record format that achieve these goals.

4.1. Encrypted SNI Record

SNI Encryption keys can be published using the following ESNIKeys structure.

```
    // Copied from TLS 1.3
    struct {
        NamedGroup group;
        opaque key_exchange<1..2^16-1>;
    } KeyShareEntry;

    struct {
        uint16 version;
        uint8 checksum[4];
        opaque public_name<1..2^16-1>;
        KeyShareEntry keys<4..2^16-1>;
        CipherSuite cipher_suites<2..2^16-2>;
        uint16 padded_length;
        uint64 not_before;
        uint64 not_after;
        Extension extensions<0..2^16-1>;
    } ESNIKeys;
```

version The version of the structure. For this specification, that value SHALL be 0xff02. Clients MUST ignore any ESNIKeys structure with a version they do not understand. [[NOTE: This means that the RFC will presumably have a nonzero value.]]

checksum The first four (4) octets of the SHA-256 message digest [RFC6234] of the ESNIKeys structure. For the purpose of computing

the checksum, the value of the "checksum" field MUST be set to zero.

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 5.1.2.

keys The list of keys which can be used by the client to encrypt the SNI. Every key being listed MUST belong to a different group.

padded_length : The length to pad the ServerNameList value to prior to encryption. This value SHOULD be set to the largest ServerNameList the server expects to support rounded up the nearest multiple of 16. If the server supports wildcard names, it SHOULD set this value to 260.

not_before The moment when the keys become valid for use. The value is represented as seconds from 00:00:00 UTC on Jan 1 1970, not including leap seconds.

not_after The moment when the keys become invalid. Uses the same unit as not_before.

extensions A list of extensions that the client can take into consideration when generating a Client Hello message. The format is defined in [RFC8446]; Section 4.2. The purpose of the field is to provide room for additional features in the future. An extension may 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 record.

The semantics of this structure are simple: any of the listed keys may be used to encrypt the SNI for the associated domain name. The cipher suite list is orthogonal to the list of keys, so each key may be used with any cipher suite. Clients MUST parse the extension list and check for unsupported mandatory extensions. If an unsupported mandatory extension is present, clients MUST reject the ESNIKeys record.

This structure is placed in the RRData section of an ESNI record as-is. Servers MAY supply multiple ESNIKeys values, either of the same or of different versions. This allows a server to support multiple versions at once. If the server does not supply any ESNIKeys values with a version known to the client, then the client MUST behave as if no ESNIKeys were found.

The name of each ESNI record MUST match the query domain name or the query domain name's canonicalized form. That is, if a client queries example.com, the ESNI Resource Record might be:

```
example.com. 60S IN ESNI "..." "..."
```

The "checksum" field provides protection against transmission errors, including those caused by intermediaries such as a DNS proxy running on a home router.

"not_before" and "not_after" fields represent the validity period of the published ESNI keys. Clients MUST NOT use ESNI keys that was covered by an invalid checksum or beyond the published period. If none of the ESNI keys values are acceptable, the client SHOULD behave as if no ESNIKeys were found.

Servers SHOULD set the Resource Record TTL small enough so that the record gets discarded by the cache before the ESNI keys reach the end of their validity period. Note that servers MAY need to retain the decryption key for some time after "not_after", and will need to consider clock skew, internal caches and the like, when selecting the "not_before" and "not_after" values.

Client MAY cache the ESNIKeys for a particular domain based on the TTL of the Resource Record, but SHOULD NOT cache it based on the not_after value, to allow servers to rotate the keys often and improve forward secrecy.

Note that the length of this structure MUST NOT exceed 2^16 - 1, as the RDLENGTH is only 16 bits [RFC1035].

4.2. Encrypted SNI DNS Resolution

This section describes a client ESNI resolution algorithm using a new "address_set" extension described below. Future specifications may introduce new extensions and corresponding resolution algorithms.

4.2.1. Address Set Extension

ESNIKeys records MAY indicate a specific IP address(es) for the host(s) in possession of the ESNI private key via the following mandatory "address_set" ESNIKeys extension:

```
    enum {
        address_set(0x1001), (65535)
    } ExtensionType;
```

The body of this extension is encoded using the following structure.

```
       enum {
           address_v4(4),
           address_v6(6),
       } AddressType;

       struct {
           AddressType address_type;
           select (address_type) {
               case address_v4: {
                   opaque ipv4Address[4];
               }
               case address_v6: {
                   opaque ipv6Address[16];
               }
           }
       } Address;

       struct {
           Address address_set<1..2^16-1>;
       } AddressSet;
```

address_set A set of Address structures containing IPv4 or IPv6 addresses to hosts which have the corresponding private ESNI key.

4.2.2. Resolution Algorithm

Clients obtain ESNI records by querying the DNS for ESNI-enabled server domains. In cases where the domain of the A or AAAA records being resolved do not match the SNI Server Name, such as when [RFC7838] is being used, the alternate domain should be used for querying the ESNI TXT record. (See Section 2.3 of [RFC7838] for more details.)

Clients SHOULD initiate ESNI queries in parallel alongside normal A or AAAA queries to obtain address information in a timely manner in the event that ESNI is available. The following algorithm describes a procedure by which clients can process ESNIKeys responses as they arrive to produce addresses for ESNI-capable hosts.

1. If an ESNIKeys response with an "address_set" extension arrives before an A or
AAAA response, clients SHOULD initiate TLS with ESNI to the provided
address(es).

2. If an A or AAAA response arrives before the ESNIKeys response, clients SHOULD wait up
to CD milliseconds before initiating TLS to either address. (Clients may begin
TCP connections in this time. QUIC connections should wait.) If an ESNIKeys
response with an "address_set" extension arrives in this time, clients SHOULD
initiate TLS with ESNI to the provided address(es). If an ESNIKeys response
without an "address_set" extension arrives in this time, clients MAY initiate
TLS with ESNI to the address(es) in the A or AAAA response. If no ESNIKeys
response
arrives in this time, clients SHOULD initiate TLS without ESNI to the available
address(es).

CD (Connection Delay) is a configurable parameter. The recommended
value is 50 milliseconds, as per the guidance in [RFC8305].

5. The "encrypted_server_name" extension

The encrypted SNI is carried in an "encrypted_server_name" extension,
defined as follows:

```
   enum {
       encrypted_server_name(0xffce), (65535)
   } ExtensionType;
```

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

```
   struct {
       CipherSuite suite;
       KeyShareEntry key_share;
       opaque record_digest<0..2^16-1>;
       opaque encrypted_sni<0..2^16-1>;
   } ClientEncryptedSNI;
```

suite The cipher suite used to encrypt the SNI.

key_share The KeyShareEntry carrying the client's public ephemeral
key shared used to derive the ESNI key.

record_digest A cryptographic hash of the ESNIKeys structure from
which the ESNI key was obtained, i.e., from the first byte of
"checksum" to the end of the structure. This hash is computed
using the hash function associated with "suite".

encrypted_sni The ClientESNIInner structure, AEAD-encrypted using
cipher suite "suite" and the key generated as described below.

For servers (in EncryptedExtensions), this extension contains the following structure:

```
   enum {
       esni_accept(0),
       esni_retry_request(1),
   } ServerESNIResponseType;

   struct {
       ServerESNIResponseType response_type;
       select (response_type) {
           case esni_accept:        uint8 nonce[16];
           case esni_retry_request: ESNIKeys retry_keys<1..2^16-1>;
       }
   } ServerEncryptedSNI;
```

response_type Indicates whether the server processed the client ESNI extension. (See Section 5.1.2 and Section 5.2.}

nonce The contents of ClientESNIInner.nonce. (See Section 5.1.)

retry_keys One or more ESNIKeys structures containing the keys that the client should use on subsequent connections to encrypt the ClientESNIInner structure.

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

```
   enum {
       esni_required(121),
   } AlertDescription;
```

Finally, requirements in Section 5.1 and Section 5.2 require implementations to track, alongside each PSK established by a previous connection, whether the connection negotiated this extension with the "esni_accept" response type. If so, this is referred to as an "ESNI PSK". Otherwise, it is a "non-ESNI PSK". This may be implemented by adding a new field to client and server session states.

5.1. Client Behavior

5.1.1. Sending an encrypted SNI

In order to send an encrypted SNI, the client MUST first select one of the server ESNIKeyShareEntry values and generate an (EC)DHE share in the matching group. This share will then be sent to the server in

the "encrypted_sni" extension and used to derive the SNI encryption key. It does not affect the (EC)DHE shared secret used in the TLS key schedule. It MUST also select an appropriate cipher suite from the list of suites offered by the server. If the client is unable to select an appropriate group or suite it SHOULD ignore that ESNIKeys value and MAY attempt to use another value provided by the server. (Recall that servers might provide multiple ESNIKeys in response to a ESNI record query.) The client MUST NOT send encrypted SNI using groups or cipher suites not advertised by the server.

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

Let Z be the DH shared secret derived from a key share in ESNIKeys and the corresponding client share in ClientEncryptedSNI.key_share. The SNI encryption key is computed from Z as follows:

```
Zx = HKDF-Extract(0, Z)
key = HKDF-Expand-Label(Zx, "esni key", Hash(ESNIContents), key_length)
iv = HKDF-Expand-Label(Zx, "esni iv", Hash(ESNIContents), iv_length)
```

where ESNIContents is as specified below and Hash is the hash function associated with the HKDF instantiation.

```
struct {
    opaque record_digest<0..2^16-1>;
    KeyShareEntry esni_key_share;
    Random client_hello_random;
} ESNIContents;
```

The client then creates a ClientESNIInner structure:

```
struct {
    ServerNameList sni;
    opaque zeros[ESNIKeys.padded_length - length(sni)];
} PaddedServerNameList;

struct {
    uint8 nonce[16];
    PaddedServerNameList realSNI;
} ClientESNIInner;
```

nonce A random 16-octet value to be echoed by the server in the "encrypted_server_name" extension.

sni The true SNI, that is, the ServerNameList that would have been sent in the plaintext "server_name" extension.

zeros Zero padding whose length makes the serialized PaddedServerNameList struct have a length equal to ESNIKeys.padded_length.

This value consists of the serialized ServerNameList from the "server_name" extension, padded with enough zeroes to make the total structure ESNIKeys.padded_length bytes long. The purpose of the padding is to prevent attackers from using the length of the "encrypted_server_name" extension to determine the true SNI. If the serialized ServerNameList is longer than ESNIKeys.padded_length, the client MUST NOT use the "encrypted_server_name" extension.

The ClientEncryptedSNI.encrypted_sni value is then computed using the usual TLS 1.3 AEAD:

```
    encrypted_sni = AEAD-Encrypt(key, iv, ClientHello.KeyShareClientHello,
ClientESNIInner)
```

Where ClientHello.KeyShareClientHello is the body of the extension but not including the extension header. Including ClientHello.KeyShareClientHello in the AAD of AEAD-Encrypt binds the ClientEncryptedSNI value to the ClientHello and prevents cut-and-paste attacks.

Note: future extensions may end up reusing the server's ESNIKeyShareEntry for other purposes within the same message (e.g., encrypting other values). Those usages MUST have their own HKDF labels to avoid reuse.

[[OPEN ISSUE: If in the future you were to reuse these keys for 0-RTT priming, then you would have to worry about potentially expanding twice of Z_extracted. We should think about how to harmonize these to make sure that we maintain key separation.]]

This value is placed in an "encrypted_server_name" extension.

The client MUST place the value of ESNIKeys.public_name in the "server_name" extension. (This is required for technical conformance with [RFC7540]; Section 9.2.) The client MUST NOT send a "cached_info" extension [RFC7924] with a CachedObject entry whose CachedInformationType is "cert".

5.1.2. Handling the server response

If the server negotiates TLS 1.3 or above and provides an "encrypted_server_name" extension in EncryptedExtensions, the client then processes the extension's "response_type" field:

- If the value is "esni_accept", the client MUST check that the extension's "nonce" field matches ClientESNIInner.nonce and otherwise abort the connection with an "illegal_parameter" alert. The client then proceeds with the connection as usual, verifying the certificate against the desired name.

- If the value is "esni_retry_request", the client proceeds with the handshake, verifying the certificate against ESNIKeys.public_name as described in Section 5.1.3. If verification 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 verified, the client MUST abort the connection with an "esni_required" alert. It then processes the "retry_keys" field from the server's "encrypted_server_name" extension.

  If one of the values contains a version supported by the client, it can regard the ESNI keys as securely replaced by the server. It SHOULD retry the handshake with a new transport connection, using that value to encrypt the SNI. 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 ESNI as securely disabled by the server. As below, it SHOULD then retry the handshake with a new transport connection and ESNI 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_server_name" extension in EncryptedExtensions, the client proceeds with the handshake, verifying the certificate against ESNIKeys.public_name as described in Section 5.1.3. The client MUST NOT enable the False Start optimization [RFC7918] for this handshake. If verification 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 ESNI.

Otherwise, when the handshake completes successfully with the public name verified, the client MUST abort the connection with an "esni_required" alert. The client can then regard ESNI as securely

disabled by the server. It SHOULD retry the handshake with a new transport connection and ESNI disabled.

[[TODO: Key replacement is significantly less scary than saying that ESNI-naive servers bounce ESNI off. Is it worth defining a strict mode toggle in the ESNI keys, for a deployment to indicate it is ready for that?]]

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

If the server sends a HelloRetryRequest in response to the ClientHello and the client can send a second updated ClientHello per the rules in [RFC8446], the "encrypted_server_name" extension values which do not depend on the (possibly updated) ClientHello.KeyShareClientHello, i.e,, ClientEncryptedSNI.suite, ClientEncryptedSNI.key_share, and ClientEncryptedSNI.record_digest, MUST NOT change across ClientHello messages. Moreover, ClientESNIInner.nonce and ClientESNIInner.realSNI MUST not change across ClientHello messages. Informally, the values of all unencrypted extension information, as well as the inner extension plaintext, must be consistent between the first and second ClientHello messages.

5.1.3. Verifying against the public name

When the server cannot decrypt or does not process the "encrypted_server_name" extension, it continues with the handshake using the cleartext "server_name" extension instead (see Section 5.2). Clients that offer ESNI then verify the certificate with the public name, as follows:

- o 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 5.1.1 requires the client to only offer ESNI-established sessions, and Section 5.2 requires the server to decline ESNI-established sessions if it did not accept ESNI.

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

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

Note that verifying a connection for the public name does not verify 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 5.1.2. This may be implemented, for instance, by reporting a failed connection with a dedicated error code.

5.2. Client-Facing Server Behavior

Upon receiving an "encrypted_server_name" 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.

If the ClientEncryptedSNI.record_digest value does not match the cryptographic hash of any known ESNIKeys structure, it MUST ignore the extension and proceed with the connection, with the following added behavior:

- o It MUST include the "encrypted_server_name" extension in EncryptedExtensions message with the "response_type" field set to "esni_retry_requested" and the "retry_keys" field set to one or more ESNIKeys structures with up-to-date keys. Servers MAY supply multiple ESNIKeys values of different versions. This allows a server to support multiple versions at once.

- o The server MUST ignore all PSK identities in the ClientHello which correspond to ESNI PSKs. ESNI PSKs offered by the client are associated with the ESNI name. The server was unable to decrypt then ESNI name, so it should not resume them when using the cleartext SNI name. This restriction allows a client to reject resumptions in Section 5.1.3.

If the ClientEncryptedSNI.record_digest value does match the cryptographic hash of a known ESNIKeys, the server performs the following checks:

- o If the ClientEncryptedSNI.key_share group does not match one in the ESNIKeys.keys, it MUST abort the connection with an "illegal_parameter" alert.

- o If the length of the "encrypted_server_name" extension is inconsistent with the advertised padding length (plus AEAD expansion) the server MAY abort the connection with an "illegal_parameter" alert without attempting to decrypt.

Assuming these checks succeed, the server then computes K_sni and decrypts the ServerName value. If decryption fails, the server MUST abort the connection with a "decrypt_error" alert.

If the decrypted value's length is different from the advertised ESNIKeys.padded_length or the padding consists of any value other than 0, then the server MUST abort the connection with an illegal_parameter alert. Otherwise, the server uses the PaddedServerNameList.sni value as if it were the "server_name" extension. Any actual "server_name" extension is ignored, which also means the server MUST NOT send the "server_name" extension to the client.

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.

5.3. Shared Mode Server Behavior

A server operating in Shared Mode uses PaddedServerNameList.sni as if it were the "server_name" extension to finish the handshake. It SHOULD pad the Certificate message, via padding at the record layer, such that its length equals the size of the largest possible Certificate (message) covered by the same ESNI key. Moreover, the server MUST include the "encrypted_server_name" extension in EncryptedExtensions with the "response_type" field set to "esni_accept" and the "nonce" field set to the decrypted PaddedServerNameList.nonce value from the client "encrypted_server_name" extension.

If the server sends a NewSessionTicket message, the corresponding ESNI PSK MUST be ignored by all other servers in the deployment when not negotiating ESNI, including servers which do not implement this specification. This may be implemented by adding a new field to the server session state which earlier implementations cannot parse.

This restriction provides robustness for rollbacks (see Section 6.1).

5.4. Split Mode Server Behavior

In Split Mode, the backend server must know PaddedServerNameList.nonce to echo it back in EncryptedExtensions and complete the handshake. Appendix A describes one mechanism for sending both PaddedServerNameList.sni and ClientESNIInner.nonce to

the backend server. Thus, backend servers function the same as servers operating in Shared Mode.

As in Shared Mode, if the backend server sends a NewSessionTicket message, the corresponding ESNI PSK MUST be ignored by other servers in the deployment when not negotiating ESNI, including servers which do not implement this specification.

6. Compatibility Issues

Unlike most TLS extensions, placing the SNI value in an ESNI 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 ESNI 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 ESNI keys and servers, at the cost of extra round-trips due to a retry. Two specific scenarios are detailed below.

6.1. Misconfiguration and Deployment Concerns

It is possible for ESNI 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 ESNI keys, or if a deployment of ESNI must be rolled back on the server.

The retry mechanism repairs most such inconsistencies. If server and advertised keys mismatch, the server will respond with esni_retry_requested. If the server does not understand the "encrypted_server_name" 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 5.1.2.

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

6.2. 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_server_name" 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 5.1.2 or result in a connection failure. A proxy which is not authoritative for the public name cannot forge a signal to disable ESNI.

A non-conformant MITM proxy which instead forwards the ESNI 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.

7. Security Considerations

7.1. Why is cleartext DNS OK?

In comparison to [I-D.kazuho-protected-sni], wherein DNS Resource Records are signed via a server private key, ESNIKeys 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 ESNIKeys (so that the client encrypts SNI to them) or strip the ESNIKeys from the response. However, in the face of an attacker that controls DNS, no SNI 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 ESNIKeys 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 SNI is desired. Moreover, as noted in the introduction, SNI encryption is less useful without encryption of DNS queries in transit via DoH or DPRIVE mechanisms.

7.2. 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 ESNI design against them.

7.2.1. Mitigate against replay attacks

Since the SNI encryption key is derived from a (EC)DH operation between the client's ephemeral and server's semi-static ESNI key, the ESNI encryption is bound to the Client Hello. It is not possible for an attacker to "cut and paste" the ESNI value in a different Client Hello, with a different ephemeral key share, as the terminating server will fail to decrypt and verify the ESNI value.

7.2.2. 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 ESNI key is provided, sharing is optimally bound by the number of hosts that share an IP address. Server operators may further limit sharing by sending different Resource Records containing ESNIKeys with different keys using a short TTL.

7.2.3. Prevent SNI-based DoS attacks

This design requires servers to decrypt ClientHello messages with ClientEncryptedSNI 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.

7.2.4. Do not stick out

As more clients enable ESNI support, e.g., as normal part of Web browser functionality, with keys supplied by shared hosting providers, the presence of ESNI extensions becomes less suspicious and part of common or predictable client behavior. In other words, if all Web browsers start using ESNI, the presence of this value does not signal suspicious behavior to passive eavesdroppers.

7.2.5. Forward secrecy

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

7.2.6. Proper security context

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

7.2.7. Split server spoofing

Assuming ESNIKeys retrieved from DNS are validated, e.g., via DNSSEC or fetched from a trusted Recursive Resolver, spoofing a server operating in Split Mode is not possible. See Section 7.1 for more details regarding cleartext DNS.

Validating the ESNIKeys structure additionally validates the public name. This validates any retry signals from the server because the client validates the server certificate against the public name before retrying.

7.2.8. 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.

7.3. Misrouting

Note that the backend server has no way of knowing what the SNI was, but that does not lead to additional privacy exposure because the backend server also only has one identity. This does, however, change the situation slightly in that the backend server might previously have checked SNI and now cannot (and an attacker can route a connection with an encrypted SNI to any backend server and the TLS connection will still complete). However, the client is still responsible for verifying the server's identity in its certificate.

[[TODO: Some more analysis needed in this case, as it is a little odd, and probably some precise rules about handling ESNI and no SNI uniformly?]]

8. IANA Considerations

8.1. Update of the TLS ExtensionType Registry

IANA is requested to create an entry, encrypted_server_name(0xffce), in the existing registry for ExtensionType (defined in [RFC8446]), with "TLS 1.3" column values being set to "CH, EE", and "Recommended" column being set to "Yes".

8.2. Update of the TLS Alert Registry

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

8.3. Update of the Resource Record (RR) TYPEs Registry

IANA is requested to create an entry, ESNI(0xff9f), in the existing registry for Resource Record (RR) TYPEs (defined in [RFC6895]) with "Meaning" column value being set to "Encrypted SNI".

9. References

9.1. Normative References

[I-D.ietf-tls-exported-authenticator]
Sullivan, N., "Exported Authenticators in TLS", draft-ietf-tls-exported-authenticator-08 (work in progress), October 2018.

[RFC1035] Mockapetris, P., "Domain names - implementation and specification", STD 13, RFC 1035, DOI 10.17487/RFC1035, November 1987, <https://www.rfc-editor.org/info/rfc1035>.

[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>.

[RFC6066] Eastlake 3rd, D., "Transport Layer Security (TLS) Extensions: Extension Definitions", RFC 6066, DOI 10.17487/RFC6066, January 2011, <https://www.rfc-editor.org/info/rfc6066>.

[RFC6234] Eastlake 3rd, D. and T. Hansen, "US Secure Hash Algorithms (SHA and SHA-based HMAC and HKDF)", RFC 6234, DOI 10.17487/RFC6234, May 2011, <https://www.rfc-editor.org/info/rfc6234>.

[RFC6895] Eastlake 3rd, D., "Domain Name System (DNS) IANA Considerations", BCP 42, RFC 6895, DOI 10.17487/RFC6895, April 2013, <https://www.rfc-editor.org/info/rfc6895>.

[RFC7540] Belshe, M., Peon, R., and M. Thomson, Ed., "Hypertext Transfer Protocol Version 2 (HTTP/2)", RFC 7540, DOI 10.17487/RFC7540, May 2015, <https://www.rfc-editor.org/info/rfc7540>.

[RFC7838] Nottingham, M., McManus, P., and J. Reschke, "HTTP Alternative Services", RFC 7838, DOI 10.17487/RFC7838, April 2016, <https://www.rfc-editor.org/info/rfc7838>.

[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>.

[RFC8305] Schinazi, D. and T. Pauly, "Happy Eyeballs Version 2: Better Connectivity Using Concurrency", RFC 8305, DOI 10.17487/RFC8305, December 2017, <https://www.rfc-editor.org/info/rfc8305>.

[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>.

9.2. Informative References

[I-D.ietf-doh-dns-over-https]
Hoffman, P. and P. McManus, "DNS Queries over HTTPS (DoH)", draft-ietf-doh-dns-over-https-14 (work in progress), August 2018.

[I-D.ietf-tls-sni-encryption]
Huitema, C. and E. Rescorla, "Issues and Requirements for SNI Encryption in TLS", draft-ietf-tls-sni-encryption-04 (work in progress), November 2018.

[I-D.kazuho-protected-sni]
Oku, K., "TLS Extensions for Protecting SNI", draft-kazuho-protected-sni-00 (work in progress), July 2017.

[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>.

Appendix A. Communicating SNI and Nonce to Backend Server

When operating in Split Mode, backend servers will not have access to PaddedServerNameList.sni or ClientESNIInner.nonce without access to the ESNI keys or a way to decrypt ClientEncryptedSNI.encrypted_sni.

One way to address this for a single connection, at the cost of having communication not be unmodified TLS 1.3, is as follows. Assume there is a shared (symmetric) key between the client-facing server and the backend server and use it to AEAD-encrypt Z and send the encrypted blob at the beginning of the connection before the ClientHello. The backend server can then decrypt ESNI to recover the true SNI and nonce.

Another way for backend servers to access the true SNI and nonce is by the client-facing server sharing the ESNI keys.

Appendix B. 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.

B.1. TLS-layer

B.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.

B.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.

B.2. Application-layer

B.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 C. 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 ESNI. 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 (as in Appendix A and thus cannot be used with an unmodified backend server.

- A conformant middlebox will strip every extension, which might result in a ClientHello which is just unacceptable to the server (more analysis needed).

Appendix D. 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 ESNI 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
Apple, Inc.

Email: cawood@apple.com