tls
Internet-Draft
Intended status: Experimental
Expires: January 3, 2019

E. Rescorla
RTFM, Inc.
K. Oku
Fastly
N. Sullivan
Cloudflare
C. Wood
Apple, Inc.
July 02, 2018

Encrypted Server Name Indication for TLS 1.3
draft-rescorla-tls-esni-00

Abstract

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

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 http://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 January 3, 2019.

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 [I-D.ietf-tls-tls13] 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:

- Cleartext client DNS queries.
- Visible server IP addresses, assuming the the server is not doing domain-based virtual hosting.
- 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

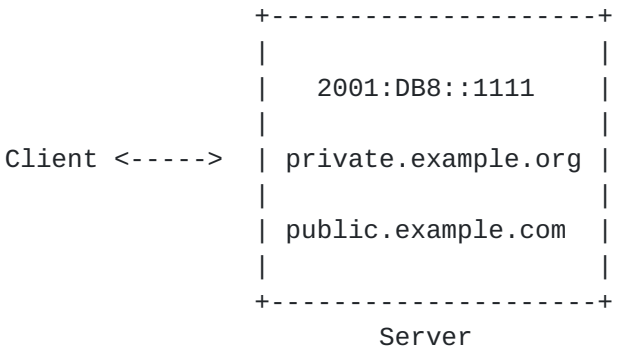

Figure 1: Shared Mode Topology

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, 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

The protocol designed in this document is quite straightforward.

First, the provider publishes a public key 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 preconfigured.

When a client wants to form a TLS connection to any of the domains served by an ESNI-supporting provider, it replaces the "server_name" extension in the ClientHello with 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

SNI Encryption keys can be published in the DNS using the ESNIKeys structure, defined below.

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

struct {
    uint8 checksum[4];
    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;
```

checksum The first four (4) octets of the SHA-256 message digest [RFC6234] of the ESNIKeys structure starting from the first octet of "keys" to the end of the structure.

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 [I-D.ietf-tls-tls13]; Section 4.2. The purpose of the field is to provide room for additional features in the future; this document does not define any extension.

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.

This structure is placed in the RRData section of a TXT record as a base64-encoded string. If this encoding exceeds the 255 octet limit of TXT strings, it must be split across multiple concatenated strings as per Section 3.1.3 of [RFC4408].

The name of each TXT record MUST match the name composed of _esni and the query domain name. That is, if a client queries example.com, the ESNI TXT Resource Record might be:

```
_esni.example.com. 60S IN TXT "..." "..."
```

Servers MUST ensure that if multiple A or AAAA records are returned for a domain with ESNI support, all the servers pointed to by those records are able to handle the keys returned as part of a ESNI TXT record for that domain.

Clients obtain these records by querying DNS for ESNI-enabled server domains. Thus, 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 "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. 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].

5. The "encrypted_server_name" extension

The encrypted SNI is carried in an "encrypted_server_name" extension, which contains an EncryptedSNI structure:

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

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".

suite The cipher suite used to encrypt the SNI.

encrypted_sni The original ServerNameList from the "server_name" extension, padded and AEAD-encrypted using cipher suite "suite" and with the key generated as described below.

5.1. Client Behavior

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 is then used for the client's "key_share" extension and will be used to derive both the SNI encryption key and the (EC)DHE shared secret which is used in the TLS key schedule. This has two important implications:

- The client MUST only provide one KeyShareEntry
- The server is committing to support every group in the ESNIKeys list (see below for server behavior).

The SNI encryption key is computed from the DH shared secret Z as follows:

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

The client then creates a PaddedServerNameList:

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

This value consists of the serialized ServerNameList 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 EncryptedSNI.encrypted_sni value is then computed using the usual TLS 1.3 AEAD:

```
encrypted_sni = AEAD-Encrypt(key, iv, "", PaddedServerNameList)
```

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 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. Similarly, if the server uses the same key for ESNI as it does in ServerKeyShare, this is going to involve re-use of Z in some hard to analyze ways. Of course, this would also involve abandoning PFS.]]

This value is placed in an "encrypted_server_name" extension.

The client MAY either omit the "server_name" extension or provide an innocuous dummy one (this is required for technical conformance with [RFC7540]; Section 9.2.)

5.2. Client-Facing Server Behavior

Upon receiving an "encrypted_server_name" extension, the client-facing server MUST first perform the following checks:

- If it is unable to negotiate TLS 1.3 or greater, it MUST abort the connection with a "handshake_failure" alert.

- If the EncryptedSNI.record_digest value does not match the cryptographic hash of any known ENSIKeys structure, it MUST abort the connection with an "illegal_parameter" alert. This is necessary to prevent downgrade attacks. [[OPEN ISSUE: We looked at ignoring the extension but concluded this was better.]]

- If more than one KeyShareEntry has been provided, or if that share's group does not match that for the SNI encryption key, it MUST abort the connection with an "illegal_parameter" alert.

- 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.

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.

5.4. Split Mode Server Behavior

The backend Server ignores both the "encrypted_server_name" and the "server_name" (if any) and completes the handshake as usual. If in Shared Mode, the server will still know the true SNI, and can use it for certificate selection. In Split Mode, it may not know the true SNI and so will generally be configured to use a single certificate. Appendix A describes a mechanism for communicating the true SNI to the backend server.

Similar to the Shared Mode behavior, the backend server in Split Mode 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.

[[OPEN ISSUE: Do we want "encrypted_server_name" in EE? It's clearer communication, but would make it so you could not operate a current TLS 1.3 server as a backend server.]]

6. Compatibility Issues

In general, this mechanism is designed only to be used with servers which have opted in, thus minimizing compatibility issues. However, there are two scenarios where that does not apply, as detailed below.

6.1. Misconfiguration

If DNS is misconfigured so that a client receives ESNI keys for a server which is not prepared to receive ESNI, then the server will ignore the "encrypted_server_name" extension, as required by [I-D.ietf-tls-tls13]; Section 4.1.2. If the servers does not require SNI, it will complete the handshake with its default certificate. Most likely, this will cause a certificate name mismatch and thus handshake failure. Clients SHOULD not fall back to cleartext SNI, because that allows a network attacker to disclose the SNI. They 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. [I-D.ietf-tls-tls13]; Section 9.3 requires that such proxies remove any extensions they do not understand. This will have one of two results when connecting to the client-facing server:

1. The handshake will fail if the client-facing server requires SNI.

2. The handshake will succeed with the client-facing server's default certificate.

A Web client client can securely detect case (2) because it will result in a connection which has an invalid identity (most likely) but which is signed by a certificate which does not chain to a publicly known trust anchor. The client can detect this case and disable ESNI while in that network configuration.

In order to enable this mechanism, client-facing servers SHOULD NOT require SNI, but rather respond with some default certificate.

A non-conformant MITM proxy will forward the ESNI extension, substituting its own KeyShare value, with the result that the client-facing server will not be able to decrypt the SNI. This causes a hard failure. Detecting this case is difficult, but clients might opt to attempt captive portal detection to see if they are in the presence of a MITM proxy, and if so disable ESNI. Hopefully, the TLS 1.3 deployment experience has cleaned out most such proxies.

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. 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 EncryptedSNI 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

By sending SNI and ESNI values (with illegitimate digests), or by sending legitimate ESNI values for and "fake" SNI values, clients do not display clear signals of ESNI intent to passive eavesdroppers. 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 RECOMMEMDED 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.

7.2.8. Supporting multiple protocols

This design has no impact on application layer protocol negotiation. It only affects 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 [I-D.ietf-tls-tls13]), with "TLS 1.3" column values being set to "CH", and "Recommended" column being set to "Yes".

9. References

9.1. Normative References

[I-D.ietf-tls-exported-authenticator]
Sullivan, N., "Exported Authenticators in TLS", draft-ietf-tls-exported-authenticator-07 (work in progress), June 2018.

[I-D.ietf-tls-tls13]
Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", draft-ietf-tls-tls13-28 (work in progress), March 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>.

[RFC4408] Wong, M. and W. Schlitt, "Sender Policy Framework (SPF) for Authorizing Use of Domains in E-Mail, Version 1", RFC 4408, DOI 10.17487/RFC4408, April 2006, <https://www.rfc-editor.org/info/rfc4408>.

[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>.

[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>.

[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>.

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-12 (work in progress), June 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-03 (work in progress), May 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 to Backend Server

As noted in Section 5.4, the backend server will generally not know the true SNI in Split Mode. It is possible for the client-facing server to communicate the true SNI to the backend server, but at the cost of having that communication not be unmodified TLS 1.3. The basic idea is to have a shared key between the client-facing server and the backend server (this can be a symmetric key) 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.

An obvious alternative here would be to have the client-facing server forward the true SNI, but that would allow the client-facing server to lie. In this design, the attacker would need to be able to find a Z which would expand into a key that would validly AEAD-encrypt a message of his choice, which should be intractable (Hand-waving alert!).

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. Acknowledgments

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 Chris Wood also provided important ideas.

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