Encrypted SNI: Privacy and Security

No Name

October 24, 2019

Contents

1 Introduction

The TLS Server Name Indication extension [**?**] is an increasingly important part of the TLS protocol. Modern TLS server deployments often offer multiple certificates behind a single IP address. The SNI helps servers choose which certificate to choose for a given connection. However, this extension also leaks the server name to any on-path observer. With recent pushes to encrypt DNS and protect names from such adversaries, the SNI extension is a privacy problem for clients. The Encrypted SNI extension [**?**] attempts to address this privacy problem by encrypting the SNI in transit. However, to date, asserting correctness and privacy properties of the protocol proved difficult.

Formal Analysis.

writeme

Contributions.

writeme

2 Encrypted SNI Overview

Encrypted SNI is a tool for hiding server names from network connections. There are several operational goals for Encrypted SNI [**?**], described below:

- Avoid widely-deployed shared secrets: One approach to the problem would be for all clients and servers to share a secret that encrypts (and decrypts) the SNI. However, any client in this set of trusted peers could then decrypt the SNI of others. Moreover, compromise of any node in possession of the secret puts all members at risk. Thus, ESNI requires public key encryption.
- Work with non-ESNI servers to avoid fallback: Without the need for fallback, ESNI is a simple ECIES-like protocol, wherein the SNI is encrypted under a public key of the service provider.
- Do not introduce extra round trips: Encrypting the SNI must not come at the cost of extra round trips. For example, one possible approach is to SNI-based certificate authentication at the application protocol layer, e.g., using HTTP/2 Secondary Certificates [**?**], after the TLS connection finishes and is authenticated with a "public name." While this may work, the extra latency cost may be prohibitively expensive for certain clients.
- Forward secrecy: Ideally, SNI encryption would have some amount of forward secrecy. However, as SNI encryption cannot introduce additional round trips, forward secrecy is not possible using public key encryption primitives such as ECIES [**?**] or HPKE [**?**]
- Prevent SNI-based DoS attacks: A consequence of using public key encryption is that servers must perform a public key operation without having validated the client. HelloRetryRequests may help dampen the effects of DoS attacks, though these come at the cost of introducing additional complexity into the protocol. See Section 2.1 for more details.
- Mitigate replay attacks: Encrypted SNI values must not be replayable from one ClientHello to another, otherwise an attacker could use an ESNI value from a victim client message in its own ClientHello.
- Support shared and split mode: Client-facing servers which use the SNI to determine the target service may not be the entity which terminates the TLS connection. ESNI should therefore support proxies which route TLS connections to backend or origin services. This suggests two possible deployment models for ESNI, referred to as shared and split mode, as described in [**?**].

2.1 Security and Privacy Goals

ESNI assumes a standard active and on-path Dolev-Yao attacker that can arbitrarily drop, tamper, replay, and forward messages from clients. Fundamentally, a TLS handshake that negotiates ESNI should leak no more information than one which did not negotiate ESNI in the presence of this adversary. This means there are at least two necessary requirements for ESNI:

1. SNI agreement: A successful TLS handshake implies agreement on the

SNI transmitted. This means, among other things, that the client authenticated the server's certificate using the SNI, and that both client and server share the same view of the SNI negotiated.

2. SNI privacy: A successful TLS handshake that negotiates ESNI does so without leaking any information about the underlying SNI. Moreover, the SNI is known only to the client and server (or any recipient of the private ESNI key). We do not require forward secrecy for the SNI encryption.

We may optionally want to hide the fact that ESNI was negotiated, as per the "do not stick out" goal. However, this is primarily only deployment concern. Furthermore, we may also want to hide the fact that a client offered ESNI in its handshake. This may be useful for clients that wish to GREASE [**?**] the extension.

2.2 Draft-04 Design Overview and Known Attacks

Figure 2 shows the ESNI design in draft-04 of the protocol.

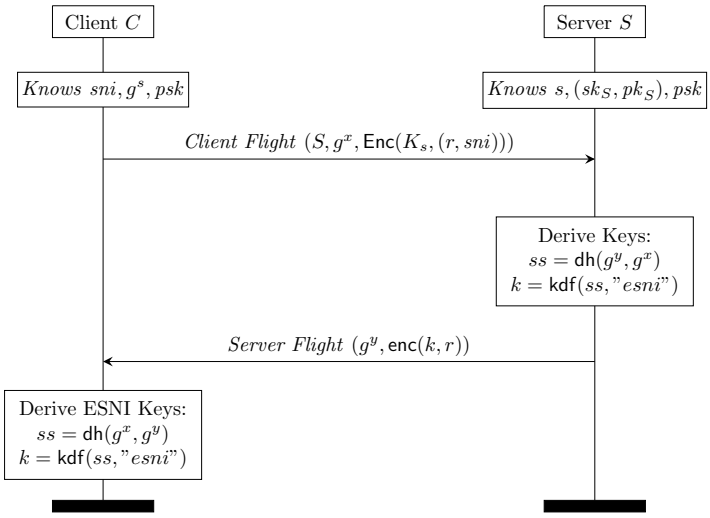

Figure 1: Simple ESNI Protocol without resumption or HelloRetryRequest support.

Early versions of ESNI did not achieve the desired security and privacy goals. For example, the first version was vulnerable to a certificate-based client reaction attack shown in Figure **??**. The SNI leak occurs if clients processed the certificate message before verifying the CertificateVerify signature. Specifically, if clients abort upon SNI mismatch between what they sent and what was received in the certificate, an attacker attempt to MITM an ESNI connection with a certificate of its choosing and try to learn the client's SNI. The core problem

was that servers did not signal to clients whether or not they processed the ESNI extension. Adding a nonce to the server's response implicitly authenticates that the server processed the SNI.

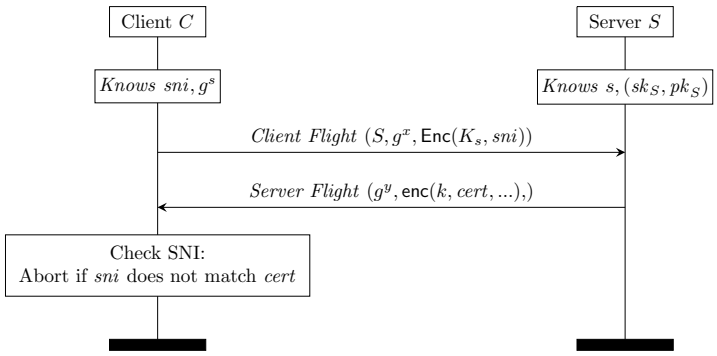

Figure 2: Simple ESNI Protocol without resumption or HelloRetryRequest support.

Despite this fix, draft-04 of ESNI does not achieve the necessary requirements stated above. We describe some more attacks on ESNI deployment configurations and the protocol below.

Probing Attacks. If an operator partitions its servers based on SNI-specific values observable on the wire, such as supported ciphersuites, key exchange algorithms, or application-layer protocols, an adversary can use these differences to learn information about the client's SNI.

HelloRetryRequest Mix and Match. In the event of a HelloRetryRequest, clients send a fresh key share and ESNI extension in the second ClientHello. Servers are expected to use the ESNI value from the same ClientHello from which it received the client key shares. However, a server may use mix and match these values, e.g., by selecting the ESNI value from the first ClientHello and the key shares from the second ClientHello.[1] This allows an on-path adversary to hijack a HelloRetryRequest, send a second ClientHello with its own key shares, and then successfully decrypt the server flight to learn the certificate.

Server Ticket Reaction Attacks. The ESNI contents are not fully bound to the entire ClientHello in draft-04. This means a ClientHello with ESNI *and* no resumption PSK can be intercepted by an on-path adversary, who then attaches a ticket and PSK binder of its choosing, and forwards the result to the original target server. If servers check whether or not the SNI obtained from the ESNI value and the ticket match and change behavior accordingly, e.g., by aborting the connection, this introduces an oracle for adversaries to learn information about client SNIs.

[1]This is not entirely farfetched, as some operators must process the SNI upon the first ClientHello in order to determine HelloRetryRequest parameters such as supported ciphersuites.

In general, the problems above seem to stem from the same problems: (1) ESNI contents are not fully bound to the ClientHello, and (2) ESNI contents are not fully bound to the rest of the TLS handshake. In the following section, we describe an ideal ESNI protocol that addresses these shortcomings.

3 Core Protocol

At its core, ESNI is a protocol between a client and server that works as described in Figure 3. It aims to provide the following guarantees:

- Client: TLS handshake secret known by entity which has the private handshake key share (y), corresponding PSK, private ESNI decryption key, and ENSI nonce.
- Server: TLS handshake secret known by entity which has the private handshake key share (x), corresponding PSK, and ENSI nonce.
- Client and server both agree on the same TLS handshake secret and transcript.

Thus, the core protocol only models the handshake up to the point of certificate receipt. It does not capture the full TLS handshake.

For presentation purposes, the core protocol uses the following helper routine:

DeriveESNIKeys(Zx)

$ek = \mathsf{kdf}(zx, '\texttt{esnikey}')$
$eiv = \mathsf{kdf}(zx, '\texttt{esniiv}')$
$ebk = \mathsf{kdf}(zx, '\texttt{esnibinder}')$
$r = \mathsf{kdf}(zx, '\texttt{esninonce}')$
return ek, eiv, ebk, r

Seal and Open are AEAD encryption and decryption functions, respectively.

Moreover, these guarantees must hold for all TLS handshake patterns, including: normal handshakes, resumption handshakes, and HelloRetryRequest handshakes. Considering the full variant, there are five secrets that influence the handshake: private signing key sk_s, secret Diffie Hellman key shares (x, y), pre-shared key(s) psk, ESNI decryption key (s), and the ESNI nonce r. Informally, we require that these values endorse each other in order to avoid the attacks described in Section 2.2.

For example, consider the ticket-based server reaction attack, wherein the ESNI nonce is not bound to a ClientHello PSK. This leads to a situation wherein a server receives a ClientHello with an attacker-controlled PSK meant for one SNI, yet an ESNI carrying an encryption of a different SNI. If said server then *checks* that these SNIs for equality and reacts differently in response, the SNI value may leak. However, if these values are properly bound, then such a check

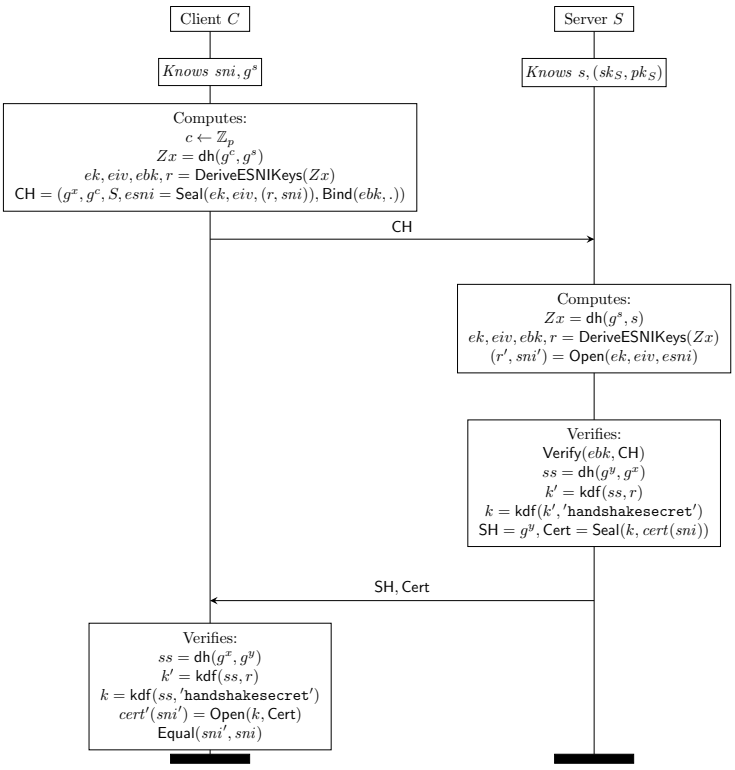

Figure 3: Minimal Core ESNI Protocol. No resumption or HelloRetryRequest support.

will not possible yield a negative answer (for honestly generated ClientHello messages), and therefore the reaction attack vanishes.

3.1 Necessary Properties

Joint binding of these secrets yields the following properties:

- Backward binding: ESNI contents are bound to the entire ClientHello such that any modification is detectable by servers. ESNI contents are *backward bound* to a ClientHello if it is not possible to modify a CH in any way without causing an ESNI check to fail.

- Forward binding: TLS handshake secrets are bound to the ESNI contents such that knowledge of both ESNI secret(s) and one of the TLS key shares is needed to derive the handshake secrets. ESNI is *forward bound* if it is not possible to learn the TLS handshake secret without both the Diffie Hellman shared secret and ESNI shared secret.

As a consequence of forward binding and the need to interoperate with ESNI-incapable servers, we also require a signalling mechanism for clients to determine whether or not the ESNI contents were used to protect the handshake secret. (Trial decryption is always an option, though other more practical solutions exist.)

Beyond these critical properties, it must also be the case that observable information, which includes messages sent on the wire and the *actions* of clients and servers using ESNI, does not leak information about the SNI. Indeed, the reaction attack described above was due in part to such an information leak.[2] We must capture this notion of information indistinguishability for completeness. We do so via *message indistinguishability* and *action indistinguishability*. Informally, message indistinguishability means that all messages written on the wire do not vary based on SNI. Similarly, action indistinguishability means that all node behavior as observed by Adv does not vary based on SNI (or any SNI-influenced value used in the protocol).

add more rigorous definition of these

describe how message indistinguishability is a property of configuration, and give the functions discussed IRL

3.2 Formal Model

In this section, we present a model for ESNI based on ProVerif [**?**, **?**]. ProVerif analyzes symbolic protocol models using processes to represent entities which communicate using messages sent over public channels. Processes can trigger security events representing attacks or critical steps of the target protocol, e.g., TLS connection establishment. Moreover, processes can save messages in lookup tables for use later on. This is useful for storing long-term keying material, such as ESNI and certificate private keys.

Our ESNI model accounts for backward and forward binding. It also accounts for action indistinguishability for client and server processes. It does not account for message indistinguishability, as this is something largely determined by configuration. Each process begins by installing or obtaining long-term secrets used each TLS connection. These secrets include public and private certificate signing keys, as well as ESNI keying material.

3.2.1 Client ESNI Process

XXX

3.2.2 Server ESNI Process

XXX

[2] Fortunately, joint binding resolved the problem by removing a branch in the way servers handle ClientHello messages.

4 Summary of Proposals

There are two proposals that conform to the core ESNI protocol described above. They are summarized below.

ESNI Proxy Transformation. A high level summary of this proposal is as follows:

- Bind the ESNI contents to the entire ClientHello with an explicit transformation function that works as follows. Given a fully-formed "private" ClientHello (with the unencrypted SNI value and ESNI nonce), encrypt the SNI value using the ClientHello as AAD, and output a "public" ClientHelo with the SNI extension replaced with the ESNI extension.
- Bind the ESNI contents to the handshake by choosing either the "public" or reconstructed "private" ClientHello to mix into the transcript.
- Never send an ESNI extension in the event of HRR.

It has the following properties:

+ XXX

– The transcript used upon ESNI negotiation is not that which is sent over the wire. This is a significant deviation from the TLS 1.3 model and introduces interesting side effects, such as the ability of an unknown third party, e.g., a backend origin server, to complete the handshake without the client being aware. (This is a problem in TLS in general, as secret information can always be exfiltrated to another party to complete the handshake.)

– Treats the handshake transcript, which is fed into the TLS key schedule as the `Info` parameter to `HKDF-Expand`, as secret information. HKDF leaks no information about this parameter if modelled as a random oracle or if the key is secret. More generally, PRFs have no guarantees about the secrecy of their inputs if the key is known. Importantly, in this proposal, an attacker does know the key in the event of HRR, yet the input is secret. Bellare and Lysyanskaya [**?**] proved that HMAC satisfies the notion of a dualPRF, which roughly states that HMAC is a PRF if either the the key or the input is secret. Thus, in practice, this should not affect security, though it may affect the proofs used for TLS 1.3.

ESNI PSK Binders and Key Schedule Injection. A high level summary of this proposal is as follows:

- Bind the ESNI contents to the entire ClientHello with an explicit PSK binder, whose value is derived from the ESNI shared secret.
- Bind the ESNI contents to the handshake by mixing the derived ESNI nonce into the key schedule.

- Always generate and send a fresh ESNI extension, even in the event of HRR.

It has the following properties:

+ Backward binding relies on existing PSK binder properties.

− Forward binding requires key schedule modification.

− In the event of session resumption, two binders are added to a ClientHello and both must be verified by the server. This goes against text in the TLS 1.3 specification [**?**], which states that servers "SHOULD NOT attempt to validate multiple binders."