Network Working Group
Internet-Draft
Intended status: Standards Track
Expires: December 22, 2017

C. Huitema
Private Octopus Inc.
E. Rescorla
RTFM, Inc.

June 20, 2017

SNI Encryption in TLS Through Tunneling
draft-huitema-tls-sni-encryption-00

Abstract

This draft describes the general problem of encryption of the Server Name Identification (SNI) parameter. The proposed solutions hide a Hidden Service behind a Fronting Service, only disclosing the SNI of the Fronting Service to external observers. The draft starts by listing known attacks against SNI encryption, and then presents two potential solutions that might mitigate these attacks. The first solution is based on TLS in TLS "quasi tunneling", and the second solution is based on "combined tickets". These solutions only require minimal extensions to the TLS protocol.

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 December 22, 2017.

Table of Contents

1. Introduction

Historically, adversaries have been able to monitor the use of web services through three channels: looking at DNS requests, looking at IP addresses in packet headers, and looking at the data stream between user and services. These channels are getting progressively closed. A growing fraction of Internet communication is encrypted, mostly using Transport Layer Security (TLS) [RFC5246]. Progressive deployment of solutions like DNS in TLS [RFC7858] mitigates the disclosure of DNS information. More and more services are colocated

on multiplexed servers, loosening the relation between IP address and web service. However, multiplexed servers rely on the Service Name Information (SNI) to direct TLS connections to the appropriate service implementation. This protocol element is transmitted in clear text. As the other methods of monitoring get blocked, monitoring focuses on the clear text SNI. The purpose of SNI encryption is to prevent that.

In the past, there have been multiple attempts at defining SNI encryption. These attempts have generally floundered, because the simple designs fail to mitigate several of the attacks listed in Section 2.

The current draft proposes two designs for SNI Encryption. Both designs hide a "Hidden Service" behind a "Fronting Service". To an external observer, the TLS connections will appear to be directed towards the Fronting Service. The cleartext SNI parameter will document the Fronting Service. A second SNI parameter will be transmitted in an encrypted form to the Fronting Service, and will allow that service to redirect the connection towards the Hidden Service.

The first design relies on tunneling TLS in TLS, as explained in Section 3. It does not require TLS extensions, but relies on conventions in the implementation of TLS 1.3 [I-D.ietf-tls-tls13] by the Client and the Fronting Server.

The second design, presented in Section 4 removes the requirement for tunneling, on simply relies on Combined Tickets. It uses the extension process for session tickets already defined in [I-D.ietf-tls-tls13].

This draft is presented as is to trigger discussions. It is expected that as the draft progresses, only one of the two proposed solutions will be retained.

1.1. Key Words

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC2119].

2. Security and Privacy Requirements for SNI Encryption

Over the past years, there have been multiple proposals to add an SNI encryption option in TLS. Many of these proposals appeared promising, but were rejected after security reviews pointed plausible attacks. In this section, we collect a list of these known attacks.

2.1. Mitigate Replay Attacks

The simplest SNI encryption designs replace in the initial TLS exchange the clear text SNI with an encrypted value, using a key known to the multiplexed server. Regardless of the encryption used, these designs can be broken by a simple replay attack, which works as follow:

1- The user starts a TLS connection to the multiplexed server, including an encrypted SNI value.

2- The adversary observes the exchange and copies the encrypted SNI parameter.

3- The adversary starts its own connection to the multiplexed server, including in its connection parameters the encrypted SNI copied from the observed exchange.

4- The multiplexed server establishes the connection to the protected service, thus revealing the identity of the service.

SNI encryption designs MUST mitigate this attack.

2.2. Avoid Widely Shared Secrets

It is easy to think of simple schemes in which the SNI is encrypted or hashed using a shared secret. This symmetric key must be known by the multiplexed server, and by every users of the protected services. Such schemes are thus very fragile, since the compromise of a single user would compromise the entire set of users and protected services.

SNI encryption designs MUST NOT rely on widely shared secrets.

2.3. Prevent SNI-based Denial of Service Attacks

Encrypting the SNI may create extra load for the multiplexed server. Adversaries may mount denial of service attacks by generating random encrypted SNI values and forcing the multiplexed server to spend resources in useless decryption attempts.

It may be argued that this is not an important DOS avenue, as regular TLS connection attempts also require the server to perform a number of cryptographic operations. However, in many cases, the SNI decryption will have to be performed by a front end component with limited resources, while the TLS operations are performed by the component dedicated to their respective services. SNI based DOS attacks could target the front end component.

SNI encryption designs MUST mitigate the risk of denial of service attacks through forced SNI decryption.

2.4. Do not stick out

In some designs, handshakes using SNI encryption can be easily differentiated from "regular" handshakes. For example, some designs require specific extensions in the Client Hello packets, or specific values of the clear text SNI parameter. If adversaries can easily detect the use of SNI encryption, they could block it, or they could flag the users of SNI encryption for special treatment.

In the future, it might be possible to assume that a large fraction of TLS handshakes use SNI encryption. If that was the case, the detection of SNI encryption would be a lesser concern. However, we have to assume that in the near future, only a small fraction of TLS connections will use SNI encryption.

SNI encryption designs MUST minimize the observable differences between the TLS handshakes that use SNI encryption and those that don't.

2.5. Forward Secrecy

The general concerns about forward secrecy apply to SNI encryption just as well as to regular TLS sessions. For example, some proposed designs rely on a public key of the multiplexed server to define the SNI encryption key. If the corresponding public key was compromised, the adversaries would be able to process archival records of past connections, and retrieve the protected SNI used in these connections. These designs failed to maintain forward secrecy of SNI encryption.

SNI encryption designs SHOULD provide forward secrecy for the protected SNI. However, this may be very hard to achieve in practice. Designs MAY compromise there, if they have other good properties.

2.6. Proper Security Context

We can design solutions in which the multiplexed server or a fronting service act as a relay to reach the protected service. Some of those solutions involve just one TLS handshake between the client and the multiplexed server, or between the client and the fronting service. The master secret is verified by verifying a certificate provided by either of these entities, but not by the protected service.

These solutions expose the client to a Man-In-The-Middle attack by the multiplexed server or by the fronting service. Even if the client has some reasonable trust in these services, the possibility of MITM attack is troubling.

The multiplexed server or the fronting services could be pressured by adversaries. By design, they could be forced to deny access to the protected service, or to divulge which client accessed it. But if MITM is possible, the adversaries would also be able to pressure them into intercepting or spoofing the communications between client and protected service.

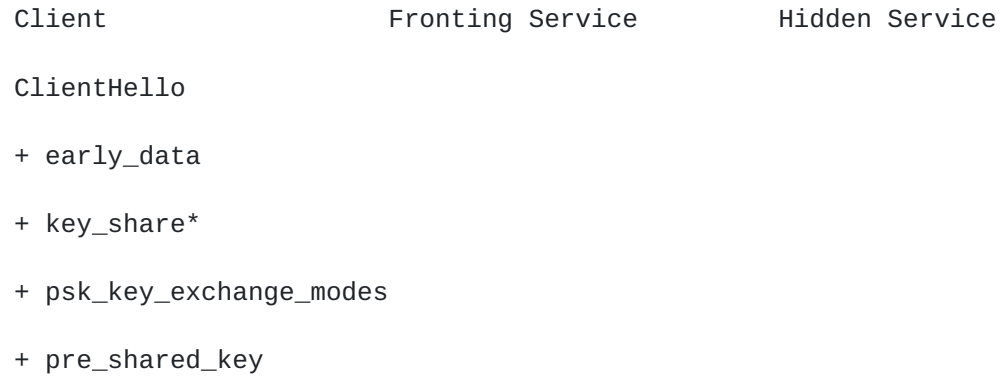
SNI encryption designs MUST ensure that the master secret are negotiated and verified "end to end", between client and protected service.

3. SNI Encapsulation Specification

We propose to provide SNI Privacy by using a form of TLS encapsulation. The big advantage of this design compared to previous attempts is that it requires effectively no changes to TLS 1.3. It only requires a way to signal to the Gateway server that the encrypted application data is actually a ClientHello which is intended for the hidden service. Once the tunneled session is established, encrypted packets will be forwarded to the Hidden Service without requiring encryption or decryption by the Fronting Service.

3.1. Tunneling TLS in TLS

The proposed design is to encapsulate a second Client Hello in the early data of a TLS connection to the Fronting Service. To the outside, it just appears that the client is resuming a session with the fronting service.



```
     + SNI = fronting
     (
      //Application data
      ClientHello#2
       + KeyShare
       + signature_algorithms*
       + psk_key_exchange_modes*
       + pre_shared_key*
       + SNI = hidden
     )
                         -------->
                              ClientHello#2
                              + KeyShare
                              + signature_algorithms*
                              + psk_key_exchange_modes*
                              + pre_shared_key*
                              + SNI = hidden  ---->

     <Application Data*>
     <end_of_early_data>    -------------------->
                                                    ServerHello
                                                +  pre_shared_key
                                                     + key_share*
                                                {EncryptedExtensions}
```

The way this works is that the Gateway decrypts the _data_ in the client's first flight, which is actually ClientHello#2 from the client, containing the true SNI and then passes it on to the Hidden server. However, the Hidden server responds with its own ServerHello which the Gateway just passes unchanged, because it's actually the response to ClientHello#2 rather than to ClientHello#1. As long as ClientHello#1 and ClientHello#2 are similar (e.g., differing only in the client's actual share (though of course it must be in the same

group)), SNI, and maybe EarlyDataIndication), then an attacker should not be able to distinguish these cases.

3.2. Tunneling design issues

The big advantage of this design is that it requires effectively no changes to TLS. It only requires a way to signal to the Fronting Server that the encrypted application data is actually a ClientHello which is intended for the hidden service.

The major disadvantage of this overall design strategy (however it's signaled) is that it's somewhat harder to implement in the co-tenanted cases than the most trivial "RealSNI" scheme. That means that it's somewhat less likely that servers will implement it "by default" and more likely that they will have to take explicit effort to allow Encrypted SNI. Conversely, however, these modes (aside from a server with a single wildcard or multi-SAN cert) involve more changes to TLS to deal with issues like "what is the server cert that is digested into the keys", and that requires more analysis, so there is an advantage to deferring that. If we have EncryptedExtensions in the client's first flight it would be possible to add RealSNI later if/when we had clearer analysis for that case.

Notes on several obvious technical issues:

1. How does the Fronting Server distinguish this case from where the initial flight is actual application data? See Section 3.2.1 for some thoughts on this.

2. Can we make this work with 0-RTT data from the client to the Hidden server? The answer is probably yes, as discussed in Section 3.2.2.

3. What happens if the Fronting Server doesn't gateway, e.g., because it has forgotten the ServerConfiguration? In that case, the client gets a handshake with the Gateway, which it will have to determine via trial decryption. At this point the Gateway supplies a ServerConfiguration and the client can reconnect as above.

4. What happens if the client does 0-RTT inside 0-RTT (as in #2 above) and the Hidden server doesn't recognize the ServerConfiguration in ClientHello#2? In this case, the client gets a 0-RTT rejection and it needs to do trial decryption to know whether the rejection was from the Gateway or the Hidden server.

The client part of that logic, including the handling of question #3 above, is discussed in Section 3.2.3.

3.2.1. Gateway logic

The big advantage of this design is that it requires effectively no changes to TLS. It only requires a way to signal to the Fronting Server that the encrypted application data is actually a ClientHello which is intended for the hidden service. The two most obvious designs are:

- Have an EncryptedExtension which indicates that the inner data is tunnelled.

- Have a "tunnelled" TLS content type.

EncryptedExtensions would be the most natural, but they were removed from the ClientHello during the TLS standardization. In Section 3.1 we assume that the second ClientHello is just transmitted as 0-RTT data, and that the servers use some form of pattern matching to differentiate between this second ClientHello and other application messages.

3.2.2. Early data

In the proposed design, the second ClientHello is sent to the Fronting Server as early data, encrypted with Client->Fronting 0-RTT key. If the Client follows the second ClientHello with 0-RTT data, that data could in theory be sent in two ways:

1. The client could use double encryption. The data is first encrypted with the Client->Hidden 0-RTT key, then wrapped and encrypted with the Client->Fronting 0-RTT key. The Fronting server would decrypt, unwrap and relay.

2. The client could just encrypt the data with the Client->Hidden 0-RTT key, and ask the server to blindly relay it.

Each of these ways has its issues. The double encryption scenario would requires two end of early data messages, one double encrypted and relayed by the Fronting Server to the Hidden Server, and another sent from Client to Fronting Server, to delimitate the end of these double encrypted stream, and also to ensure that the stream of messages is not distinguishable from simply sending 0-RTT data to the Fronting server. The blind relaying is simpler, and is the scenario described in the diagram of Section 3.1. In that scenario, the Fronting server switches to relaying mode immediately after unwrapping and forwarding the second ClientHello.

3.2.3. Client requirements

In order to use the tunneling service, the client needs to identify the Fronting Service willing to tunnel to the Hidden Service. We can assume that the client will learn the identity of suitable Fronting Services from the Hidden Service itself.

In order to tunnel the second ClientHello as 0-RTT data, the client needs to have a shared secret with the Fronting Service. To avoid the trap of "well known shared secrets" described in Section 2.2, this should be a pair wise secret. The most practical solution is to use a session resumption ticket. This requires that prior to the tunneling attempt, the client establishes regular connections with the fronting service and obtains one or several session resumption tickets.

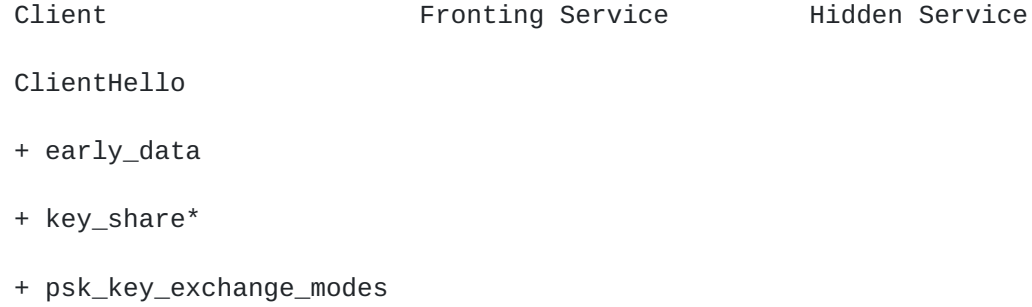
4. SNI encryption with combined tickets

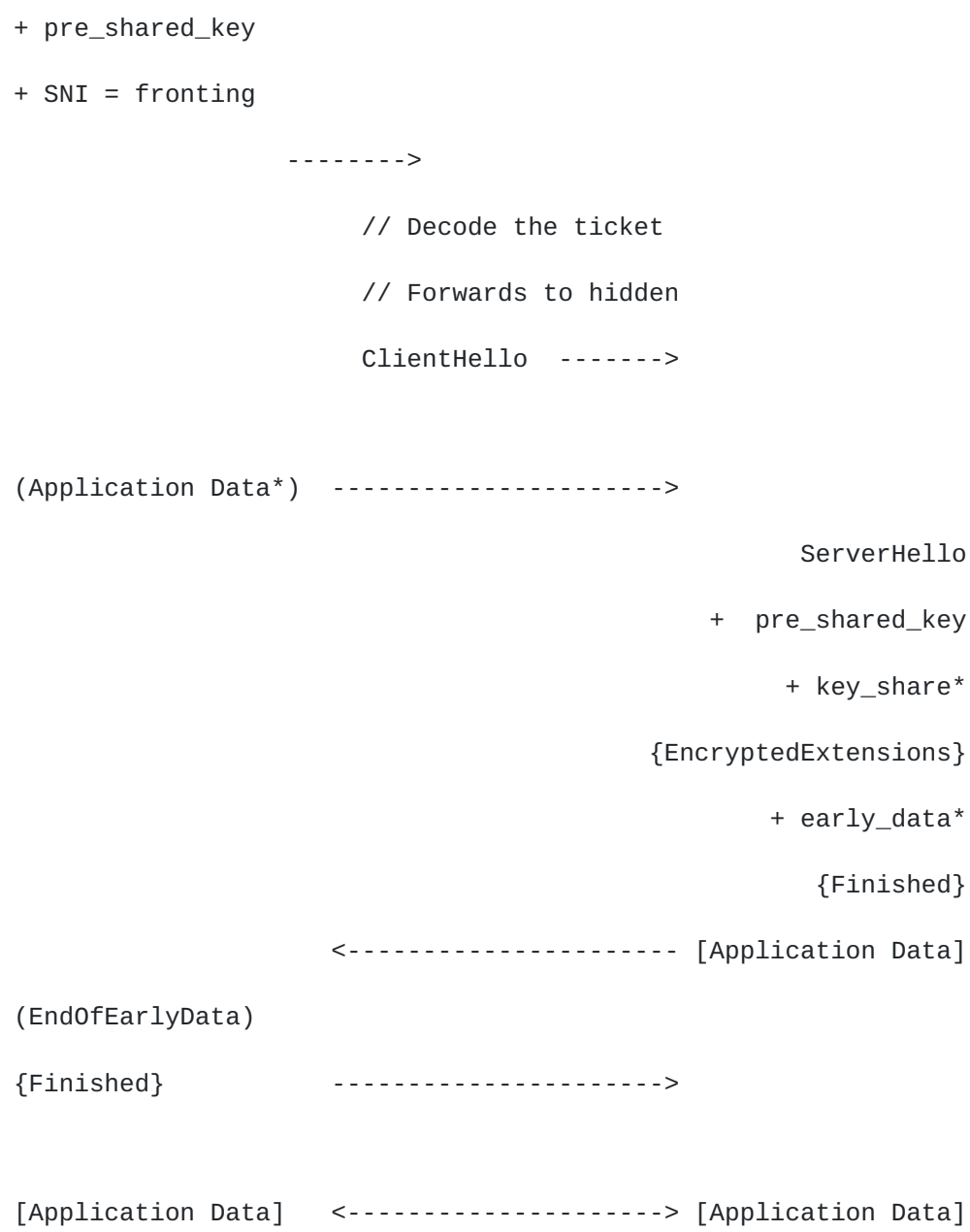
EDITOR'S NOTE: This section is an alternative design to Section 3. As the draft progresses, only one of the alternatives will be selected, and the text corresponding to the other alternative will be deleted.

We propose to provide SNI Privacy by relying solely on "combined tickets". The big advantage of this design compared to previous attempts is that it requires only minimal changes to implementations of TLS 1.3. These changes are confined to the handling of the combined ticket by Fronting and Hidden service, and to the signaling of the Fronting SNI to the client by the Hidden service.

4.1. Session resumption with combined tickets

In this example, the client obtains a combined session resumption ticket during a previous connection to the hidden service, and has learned the SNI of the fronting service. The session resumption will happen as follow:





\+ Indicates noteworthy extensions sent in the previously noted message.

* Indicates optional or situation-dependent messages/extensions that are not always sent.

() encrypted with Client->Hidden 0-RTT key

{} encrypted with Client->Hidden 1-RTT handshake

[] encrypted with Client->Hidden 1-RTT key

The Fronting server that receives the Client Hello will find the combined ticket in the pre_shared_key extensions, just as it would in a regular session resumption attempt. When parsing the ticket, the Fronting server will discover that the session really is meant to be resumed with the Hidden server. It will arrange for all the connection data to be forwarded to the Hidden server, including forwarding a copy of the initial Client Hello.

The Hidden server will receive the Client Hello. It will obtain the identity of the Fronting service from the SNI parameter. It will then parse the session resumption ticket, and proceed with the resumption of the session.

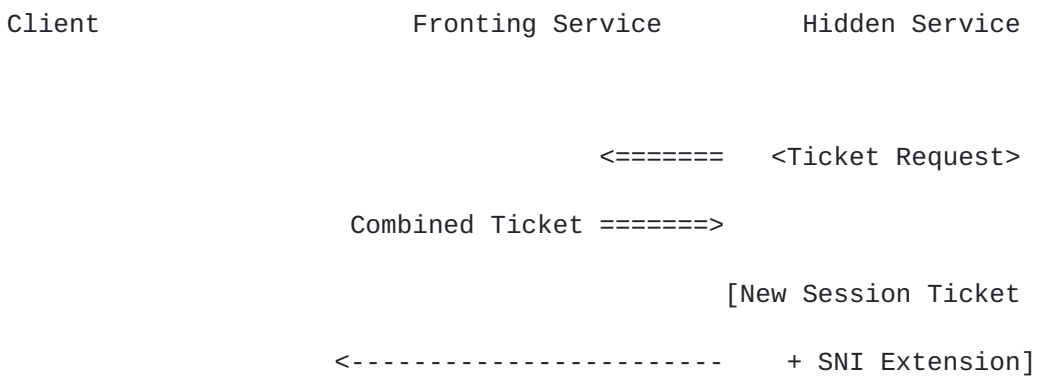
In this design, the Client Hello message is relayed unchanged from Fronting server to hidden server. This ensures that code changes are confined to the interpretation of the message parameters. The construction of handshake contexts is left unchanged.

4.2. New Combined Session Ticket

In normal TLS 1.3 operations, the server can send New Session Ticket messages at any time after the receiving the Client Finished message. The ticket structure is defined in TLS 1.3 as:

```
struct {
    uint32 ticket_lifetime;
    uint32 ticket_age_add;
    opaque ticket_nonce<1..255>;
    opaque ticket<1..2^16-1>;
    Extension extensions<0..2^16-2>;
} NewSessionTicket;
```

When SNI encryption is enabled, tickets will carry a "Fronting SNI" extension, and the ticket value itself will be negotiated between Fronting Service and Hidden Service, as in:



<==> sent on connection between Hidden and Fronting service

<> encrypted with Fronting<->Hidden key

[] encrypted with Client->Hidden 1-RTT key

In theory, the actual format of the ticket could be set by mutual agreement between Fronting Service and Hidden Service. In practice, it is probably better to provide guidance, as the ticket must meet three of requirements:

- The Fronting Server must understand enough of the combined ticket to relay the connection towards the Hidden Server;
- The Hidden Server must understand enough of the combined ticket to resume the session with the client;
- Third parties must not be able to deduce the name of the Hidden Service from the value of the ticket.

There are two plausible designs, a stateful design and an shared key design. (There is also a design in which the Hidden Server encrypts the tickets with the public key of the Fronting Server, but that does not seem very practical.) In the stateful design, the ticket are just random numbers that the Fronting server associates with the Hidden server, and the Hidden server associates with the session context. The shared key design would work as follow:

- the hidden server and the fronting server share a symmetric key K_sni.

- the "clear text" ticket includes a nonce, the ordinary ticket used for session resumption by the hidden service, and the id of the Hidden service for the Fronting Service.

- the ticket will be encrypted with AEAD, using the nonce as an IV.

- When the client reconnects to the fronting server, it decrypts the ticket using K_sni and if it succeeds, then it just forwards the CH to the hidden server indicated in id-hidden-service (which of course has to know to ignore SNI). Otherwise, it terminates the connection itself with its own SNI.

The hidden server can just refresh the ticket any time it pleases, as usual.

This design allows the Hidden Service to hides behind many Fronting Services, each using a different key. The Client Hello received by the Hidden Server carries the SNI of the Fringting Service, which the Hidden Server can use to select the appropriate K_sni.

4.3. First session

The previous sections present how sessions can be resumed with the combined ticket. Clients have that have never contacted the Hidden Server will need to obtain a first ticket during a first session. The most plausible option is to have the client directly connects to the Hidden Service, and then asks for a combined ticket. The obvious issue is that the SNI will not be encrypted for this first connection, which exposes clients to surveillance and censorship.

The client may also learn about the relation between Fronting Service and Hidden Service through an out of band channel, such as DNS service, or word of mouth. However, it is difficult to establish a combined ticket completely out of band, since the ticket must be associated to two shared secrets, one shared with the Fronting service so the second Client Hello can be sent as 0-RTT data, and the other shared with the Hidden service to ensure protection against replay attacks.

An alternative may be to use the TLS-in-TLS service described in Section 3.1 for the first contact. There will be some overhead due to tunnelling, but as we discussed in Section 3.2.3 the tunneling solution allows for safe first contact.

5. Security Considerations

The encapsulation protocol proposed in this draft mitigates the known attacks listed in Section 2. For example, the encapsulation design uses pairwise security contexts, and is not dependent on the widely shared secrets described in Section 2.2. The design also does not rely on additional public key operations by the multiplexed server or by the fronting server, and thus does not open the attack surface for denial of service discussed in Section 2.3. The session keys are negotiated end to end between the client and the protected service, as required in Section 2.6.

The combined ticket solution also mitigates the known attacks. The design also uses pairwise security contexts, and is not dependent on the widely shared secrets described in Section 2.2. The design also does not rely on additional public key operations by the multiplexed server or by the fronting server, and thus does not open the attack surface for denial of service discussed in Section 2.3. The session keys are negotiated end to end between the client and the protected service, as required in Section 2.6.

However, in some cases, proper mitigation depends on careful implementation.

5.1. Replay attacks and side channels

Both solutions mitigate the replay attacks described in Section 2.1 because adversaries cannot receive the replies intended for the client. However, the connection from the fronting service to the hidden service can be observed through side channels.

To give an obvious example, suppose that the fronting service merely relays the data by establishing a TCP connection to the hidden service. Adversaries can associate the arrival of an encrypted message to the fronting service and the setting of a connection to the hidden service, and deduce which hidden service the user accessed.

The mitigation of this attack relies on proper implementation of the fronting service. This may require cooperation from the multiplexed server.

5.2. Sticking out

The TLS encapsulation protocol mostly fulfills the requirements to "not stick out" expressed in Section 2.4. The initial messages will be sent as 0-RTT data, and will be encrypted using the 0-RTT key negotiated with the fronting service. Adversaries cannot tell

whether the client is using TLS encapsulation or some other 0-RTT service. However, this is only true if the fronting service regularly uses 0-RTT data.

The combined token solution almost perfectly fulfills the requirements to "not stick out" expressed in Section 2.4, as the observable flow of message is almost exactly the same as a regular TLS connection. However, adversaries could observe the values of the PSK Identifier that contains the combined ticket. The proposed ticket structure is designed to thwart analysis of the ticket, but if implementations are not careful the size of the combined ticket can be used as a side channel allowing adversaries to distinguish between different Hidden Services located behind the same Fronting Service.

5.3. Forward Secrecy

In the TLS encapsulation protocol, the encapsulated Client Hello is encrypted using the session resumption key. If this key is revealed, the Client Hello data will also be revealed. The mitigation there is to not use the same session resumption key multiple time.

The most common implementations of TLS tickets have the server using Session Ticket Encryption Keys (STEKs) to create an encrypted copy of the session parameters which is then stored by the client. When the client resumes, it supplies this encrypted copy, the server decrypts it, and has the parameters it needs to resume. The server need only remember the STEK. If a STEK is disclosed to an adversary, then all of the data encrypted by sessions protected by the STEK may be decrypted by an adversary.

To mitigate this attack, server implementations of the TLS encapsulation protocol SHOULD use stateful tickets instead of STEK protected TLS tickets. If they do rely on STEK protected tickets, they MUST ensure that the K_sni keys used to encrypt these tickets are rotated frequently.

6. IANA Considerations

Do we need to register an extension point? Or is it just OK to use early data?

7. Acknowledgements

A large part of this draft originates in discussion of SNI encryption on the TLS WG mailing list, including comments after the tunneling approach was first proposed in a message to that list: <https://mailarchive.ietf.org/arch/msg/tls/tXvdcqnogZgqmdfCugrV8M90Ftw>.

During the discussion of SNI Encryption in Yokohama, Deb Cooley argued that rather than messing with TLS to allow SNI encryption, we should just tunnel TLS in TLS. A number of people objected to this on the grounds of the performance cost for the gateway because it has to encrypt and decrypt everything.

After the meeting, Martin Thomson suggested a modification to the tunnelling proposal that removes this cost. The key observation is that if we think of the 0-RTT flight as a separate message attached to the handshake, then we can tunnel a second first flight in it.

The combined ticket approach was first proposed by Cedric Fournet and Antoine Delignaut-Lavaud.

8. References

8.1. Normative References

[I-D.ietf-tls-tls13]
Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", draft-ietf-tls-tls13-20 (work in progress), April 2017.

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

8.2. Informative References

[RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, DOI 10.17487/RFC5246, August 2008, <http://www.rfc-editor.org/info/rfc5246>.

[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, <http://www.rfc-editor.org/info/rfc7858>.

Authors' Addresses

Christian Huitema
Private Octopus Inc.
Friday Harbor WA 98250
U.S.A

Email: huitema@huitema.net

Eric Rescorla
RTFM, Inc.
U.S.A

Email: ekr@rtfm.com