Network Working Group
Internet-Draft
Intended status: Standards Track
Expires: April 21, 2016

K. Cairns
Washington State University
J. Mattsson
R. Skog
D. Migault
Ericsson

October 19, 2015

Session Key Interface (SKI) for TLS and DTLS
draft-cairns-tls-session-key-interface-01

Abstract

This document describes a session key interface that can be used for TLS and DTLS. The Heartbleed attack has clearly illustrated the security problems with storing private keys in the memory of the TLS server. Hardware Security Modules (HSM) offer better protection but are inflexible, especially as more (D)TLS servers are running on virtualized servers in data centers.

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 April 21, 2016.

Table of Contents

1. Introduction

Transport Layer Security (TLS) is specified in [RFC5246] and the Datagram Transport Layer Security (DTLS), which is based on TLS, is specified in [RFC6347]. During the TLS handshake, the TLS client and the TLS server exchange a symmetric session key called the premaster secret. From the premaster secret, the client random, and the server random, the endpoints derive a master secret, which in turn is used to derive the traffic encryption keys and IVs. The TLS server is authenticated during this process by presenting a certificate and

then proving possession of the private key corresponding to the public key in the certificate.

An important principle in designing security architectures is to limit access to keying material, especially long-lived secrets such as private keys. The Heartbleed attack [HEART] has illustrated the dangers of storing private keys in the memory of the TLS server.

The TLS Session Key Interface (SKI) defined in this document makes it possible to store private keys in a highly trusted key server, physically separated from client facing servers. With TLS SKI (see Figure 1), the TLS Server is split into two distinct entities called Edge Server and Key Server that communicate over an encrypted and mutually authenticated channel using e.g. TLS. The Edge Server can be placed close to the clients, reducing latency, while the Key Server is placed in a safe location. One important use case is an origin that operates a number of distributed HTTPS servers. The public certificates (not private keys) are pre-provisioned in the Edge Server. The Key Server handles all the private key operations. It retains control of the private keys and can at any time reject a request from the Edge Server, e.g. if there is reason to suspect that the Edge Server has been compromised.

The interface SKI uses modern web technologies like JSON, CBOR, HTTP, CoAP, TLS, and REST. SKI supports the most commonly used key exchange methods DHE_RSA, ECDHE_ECDSA, ECDHE_RSA, and RSA, together with X.509 [RFC5280] or raw public key [RFC7250] authentication. It does not work with PSK or SRP authentication. Even though the industry is quickly moving towards the more secure ECDHE key exchange methods, which provides perfect forward secrecy, static RSA still needs be supported in many deployments.

The remaining of the document is as follows. Section 2 defines the terms used in this document. Section 3 describes the problem statement and the need to centralize the private key operations to a centralized Key Server as well as a standard interface to interoperate with the Key Server. The resulting architecture is detailed in Section 4 followed by a security analysis and security requirements the different components as well as the SKI interface MUST meet. Section 5 describes the SKI and defines a specific SKI implementation based on HTTP and JSON. Section 6 position the SKI toward the different TLS extensions, and Section 7 illustrates the described SKI with examples.

2. Terminology

TLS Client

TLS Server

Edge Server

Key Server

SKI

3. Problem Statement

With TLS, a TLS Client can set up an authenticated and encrypted channel with a TLS Server. Authentication of the TLS Server as well as the negotiation of the TLS Session Keys are performed during the TLS hand shake. The TLS hand shake as described in [RFC5246] details two methods: RSA and ephemeral Diffie Hellman. In both case, the TLS Server is expected to perform some cryptographic operations based on a private key and thus requires to have access to the private key. When a single server is involved, the key is expected to be hosted by the server. However, numerous web applications cannot be hosted by a single TLS Server. Most of the time multiple TLS Servers are needed. In addition, multiple cloud provider or hosting providers provides resource elasticity by instantiating TLS Servers and placing these servers at the edge of the network in order to address the demand and reduce latency. The various instances of TLS Server may be inside a singe domain or across multiple domains like a private cloud combined with other third party cloud providers.

As each instance of the TLS Server needs to be able to perform some cryptographic operation with the private key, a number of ways may be envisioned:

- 1) The cryptographic material, e.g. the private key is shared between all TLS Server instances

    - a) Cryptographic material is copied into the various instances of the TLS Server

    - b) Cryptographic material is outsourced and accessed by all instances of TLS Servers

- 2) The cryptographic material is not shared and each instance has its own cryptographic material

At first, hosting private key in memory of the TLS Server exposes the cryptographic material to leakage as illustrated by the Heartbleed attack [HEART]. One common practice used to protect keys is to delegate the private key operations to a separate entity such as a Hardware Security Module (HSM), something that is supported in many TLS libraries. HSMs provide good security but are inflexible and may be difficult to deploy when the TLS server runs on a virtualized machine in the cloud, especially if the application server that uses TLS moves between different data centers. Furthermore, while HSMs protect against extraction of the private key, they do not protect against misuse in case an adversary gains possession of the HSM itself. In fact, an attacker taking control of the HSM can use the HSM to encrypt (resp. decrypt) any clear text (resp. encrypted text). Similarly, the use of a network-attached HSM does not prevent a corrupted client to have provide the full access to encryption / decryption unless some control access is performed to the data provided. In general, access control policies on the data encrypted / decrypted by the HSM are not provided. In addition, communication protocols of HSM are specific HSM vendor. There are several other proprietary session key interfaces deployed but no standardized solution.

Then, copying private keys in multiple instances increases the surface of attack is even increases the surface of attack with the number of instances of TLS Server. One way to limit the surface of attack is to use a public / private key generated for each instance of TLS Server. More specifically, when a TLS Server instance is corrupted, the and the attacker get access to the private key, this key cannot be used for another instance. However, splitting keys per instance comes also with some additional drawbacks. For example, session resumption does not work between multiple instances of TLS Servers. In addition, all newly generated public keys of each TLS Servers needs to be signed by the Certificate Authority, which comes with an additional management overhead.

The proposed TLS Session Key Interface Architecture proposes to have a common cryptographic material hold by the Key Server shared by all instances of the TLS Servers. In addition, the interface between the TLS Severs and the Key Server is limited enforced to strong access control policies so to limit the scope of use of the encryption / decryption capabilities of the Server Key.

4. TLS Session Key Interface Architecture

4.1. Architecture Overview

The TLS Session Key Interface Architecture is composed of three main components as described in Figure 1:

TLS Client are typically all web browsers or any TLS Client initiating an handshake with the TLS Server.

Edge Server are the TLS Server part seen by the TLS Client. It is designated as an Edge Server as it does not host the private key of the TLS Server. Instead, when the private key is involved, the cryptographic operation is performed by the Key Server. Edge Servers are expected to be placed close to the TLS Client in order to reduce the latency.

Key Server hosts the private key and performs the cryptographic operations on behalf of the Edge Server. Note that the Key Server may be connected to a HSM for example. In addition, they may be a single Key Server or multiple Key Servers.

In order to implement the SKI, the servers implementations and TLS libraries should make private key operation non blocking.

Figure 1: TLS Session Key Interface Architecture

4.2. Security Analysis

4.2.1. Edge Server

Edge Servers are serving the TLS traffic of the TLS Clients. Edge Servers performs all necessary operations except the cryptographic operations involving the private keys associated to the TLS Server.

If an Edge Server becomes compromised, an attacker is still likely to perform some operations with the private key of the TLS Server by interacting with the Key Server. The corrupted Edge Server may, for example, generate the TLS master secrets and impersonates the Edge Server. However, such attacks are not different from those that existed on TLS Server.

The presented architecture presents the following advantages. First the private key remains protected and cannot be retrieved by the attacker. This was obviously not the case when the key was hosted on the corrupted TLS Server. Then, the attack is contained to the

communications involving the Edge Servers. The corrupted Edge Server does not compromise the other Edge Servers in the same way as when the private key of the TLS Server is copied on all Edge Servers. With the presented architecture, addressing the attack locally to the corrupted Edge Server is sufficient. Note that in the case Edge Servers are dynamically provisioned, it is likely that the vulnerability found on one Edge Server may be also be found on other Edge Servers. Such consideration are out of scope of the proposed architecture, and are inherent to deployment cloning VMs or instantiating VMs with an identical configuration. At last, Edge Servers are not working on their own and still require some communications with a centralized Key Server. Such communications with the Key Server may also be used to qualify the activities of the Edge Servers, and thus used to detect any abnormal behaviors. This of course requires the Key Server to log and monitor the Edge Servers' activities.

If an Edge Server becomes compromised, an attacker may perform attacks such as chosen plain text attacks if it can request clear text data to be encrypted or chosen cipher text attacks in case it can provide encrypted data and get the corresponding clear text. One way to limit such attacks is to monitor the activity of the Edge Servers, and raise an alarm when suspicious activity has been detected. In case the Edge Server has been tagged with a suspicious activity, further investigations and audit may be performed on line if the Edge Server is still running or off line otherwise. One way to increase the difficulty of performing such attack is to make the chosen text harder. This could be handled at the API level for example, as detailed in Section 4.2.3.

A similar attack may be performed in an orchestrated way, for example when multiple Edge Servers are compromised and are collaborating. Collaboration may be used to perform a chosen plain text attack or a chosen cipher text attack for example. The advantage of using multiple compromised Edge Servers, is that the various requests are less likely to be detected than if being sent by a single Edge Server. Such attacks may be detected by monitoring the traffic not on a per-Edge Server basis, but instead globally, and for example look at the randomness distribution of the provided clear text or cipher text.

If a Edge Server has been compromised and its private key has be retrieved by the attacker, the attacker, is then able to send request to the Key Server on behalf of the Edge Server. If the credentials are not bound to the IP addresses, the queries attack may even be performed from another host or IP address than the Edge Server.

4.2.2. Key Server

The Key Server is a crucial element of the architecture which centralizes all the cryptographic operations involving the private key of the TLS Server. The responsibility of the Key Server is top keep the private key secret, while keeping the service available.

Although the Figure 1 represents only one Key Server, the architecture may have multiple Key Servers in order to address the traffic load or in order to provide high availability. Increasing the number of Key Servers increases the surface of attack and so the risk of leakage for the private key.

Even though the number of Key Servers may increase it number is expected to remain way below the number of Edge Servers of TLS Servers with a copy of the private key. As a result, the risks are still reduced by several orders of magnitudes.

Increasing the number may also require some coordinated monitoring. In fact, a single Key Server provides some centralized way to control the cryptographic operations requested globally and for each individual Edge Server. With multiple Key Servers, such analyze may not be performed solely within the Key Server. Instead, logging data may be outsourced to another component that performs the analysis.

If Key Server becomes compromised, the attacker is able to decrypt any cypher text encrypted with the public key. More especially, an attacker is able to read the server and client randoms as well as the pre-master secret and then generate the session key. This is true for on path traffic, but also for recorded traffic. For that purpose it is recommended to favor key exchanges that enforce perfect forward secrecy. In other words RSA is not recommended as specified in section F.1.1.2 of [RFC5246].

Key Servers centralize all cryptographic operations performed with the private key of the TLS Servers. This provides the Key Servers a bottle neck position. If the Key Servers undergo a DoS or DDoS attack, they can prevent the Edge Servers to set TLS sessions. Key Servers should be over provisioned, and should be able to rate limit requests from Edge Servers. In addition to authenticated traffic, the Edge Server should be able to detect when traffic is being replayed or when the identity of an Edge Server has been usurped - like the Edge Server being stolen its private key.

4.2.3. Communication and SKI

The communication using the SKI MUST be mutually authenticated and encrypted in order to have a malicious node hijacking the communication or pretending its is a legitimate Edge Server and serving TLS Clients.

Similarly, the communication between Edge Servers and Key Server should be encrypted in order to avoid a malicious nodes to collect a collection of clear text with their associated encrypted text and eventually perform a replay attack.

TLS or IPsec are good candidates too secure the SKI communication.

SKI MUST be designed with strong access control in order to limit the scope of actions performed by an authorized Edge Servers. This may be performed by checking the properties of the inputs as well as defining which inputs and actions are permitted.

Inputs provided to the Key Servers should be considered in order to reduce the surface of attack. Suppose the Edge Server needs to encrypt the hash of two random number. One way could do is first let the Edge Server hash the two random number and then ask the Key Server to encrypt the resulting hash. Such design exposes the Key Server to clear text attack as, any fixed value value could be fixed to the hash. On the other hand, the Edge Server could also provide the two numbers to the Key Server, which in turn perform the hash followed by the encryption. Doing so provides less control to the Edge Server for choosing the clear text. Note also that in the second case, the Key Server is performing more operations, and communications may involve more data to be carried. As a result, security and performance may be balanced.

Similarly, parameters provided should be strictly controlled in order to narrow the scope of clear text / cipher text chosen attacks, and when possible, length or syntax should be checked. In addition, when an error occurs, the Key Server should limit the information provided to the Edge Server. For example, it may be better to simply reject the request with a general error message that does not specify the specific error encountered, so this information may not be used by the attacker. On the other hand, th error should be logged precisely, so it may be used during the analysis.

4.3. Security Requirements

Here are the following requirements or recommendations regarding the architecture.

REQ 1: The activity of the Edge Servers MUST be logged and audited in order to detect suspicious activity.

REQ 2: The request from Edge Servers MUST be globally monitored in order to detect some orchestrated attacks not detected at the Edge Server level.

REQ 3: RSA based authentication is not recommended to preserve TLS Client privacy and confidentiality in case of Key Server leakage.

REQ 4: The communication between the Edge Server and the Key Server MUST be mutually authenticated and encrypted. The use of perfect forward secrecy cypher suites is recommended.

REQ 5: SKI MUST be designed to limit the possible operations performed by the Edge Server. This involves strict control of the parameters as well as specific design to avoid clear text or cipher attacks.

REQ 6: SKI MUST NOT provide the Edge Server extra information in case an error occurs.

REQ 7: SKI and Key Server MUST be monitored and logged to enable further investigation and analysis.

5. Session Key Interface (SKI)

TLS provides different methods in order to agree on the pre_master secret. One way - designated as "rsa" in [RFC5246] - consists in the TLS Client provides the pre_master secret encrypted in a Client Key Exchange message. The TLS Client encrypts the pre_master with the public key previously provided by the server in a Server Certificate message.

Other methods are based on the Diffie Hellman approach, which provides perfect forward secrecy. As described in section F.1.1.3 of [RFC4346], the TLS Server can either provide fixed Diffie Hellman parameters in a Server Certificate message or provide ephemeral Diffie Hellman parameters. In the first case, the TLS Client may authenticate the Server Certificate with a DSA, RSA, ECDSA signature. The TLS Server provides certificates the TLS Client is able to check. In other words, the signature uses hash function and signature algorithms supported by the TLS Client. When Diffie Hellman is not authenticated, then the Diffie Hellman value is not provided in the Server Certificate message. Instead, it is provided in an additional Key Server Exchange message. In the second case, when ephemeral Diffie Hellman values are provided the value is embedded in a Key

Server Exchange message with an additional Signature structure. The Signature is computed by the TLS Server over the hash of the ephemeral Diffie Hellman key together with a set of temporary values (the ClientHello.random and the ServerHello.random) to avoid replay attacks. The TLS Server provides the signature in accordance to the hash and signature function supported by the TLS Client as well as the key provided by the TLS Server in the Certificate message.

As a result, the private key of the TLS Server is only involved when the following key exchanges algorithm (KeyExchangeAlgorithm) are agreed between the TLS Client and the Edge Server:

RSA when the pre_master is entirely generated by the TLS_Client and encrypted by the TLS Client in a Client Key Exchange message. This authentication method is defined in [RFC5246].

DHE_RSA when the hash of the ephemeral Diffie Hellman key associated to the temporary values is signed with the RSA private key. This is defined in [RFC5246].

ECDHE_RSA Similar as above but with Elliptic Curve Diffie Hellman values with an RSA signature. This method is defined in [RFC4492].

ECDHE_ECDSA Similar as above but with elliptic curve signature. This method is defined in [RFC4492].

The following document only considers these key exchange protocols. If another key exchange protocol is negotiated, as currently defined, there is no need to perform cryptographic operations involving the private key. As a result, such key exchange protocols do not require the Edge Server to interact with the Key Server, and are not considered in this document. Instead, Edge Server should be provisioned with the appropriated certificates.

DISCUSSION: It is not clear to me why DHE_DSS does not sign the DHParameters.

This section designs the SKI. Section 5.1 provides an overview of the SKI. More specifically, it describes the information that is communicated between the Edge Server and the Key Server, but does not provide any details on the protocols used to exchange these information, nor how the private key is being identified. This is left to Section 5.2 provides a specific implementation based on JSON and HTTP.

5.1. SKI Protocol Overview

This section describes the interactions between the TLS Client, the Edge Server and the Key Server when either RSA or ephemeral Diffie Hellman (DHE_RSA, ECDHE_RSA or ECDHE_ECDSA) key agreement have been agreed between the TLS Client an dthe Edge Server.

The description of this section applies for TLS 1.0 [RFC2246], TLS 1.1 [RFC4346], TLS 1.2 [RFC5246], DTLS 1.0 [RFC4347], DTLS 1.1 [RFC4347] and DTLS 1.2 [RFC6347].

5.1.1. RSA

In TLS1.2 [RFC5246] every session has a "master_secret" generated from a pre_master. [RFC5246] and [RFC7627] defines different ways to generate the master_secret from the pre_master. However, the way the pre_master is agreed remains similar.

For information, in [RFC5246], the master_secret is generated as follows:

```
   master_secret = PRF(pre_master_secret, "master secret",
                       ClientHello.random + ServerHello.random)
                       [0..47];

   where:
   struct {
               uint32 gmt_unix_time;   # 4 bytes
               opaque random_bytes[28];
           } Random;
```

master_secret

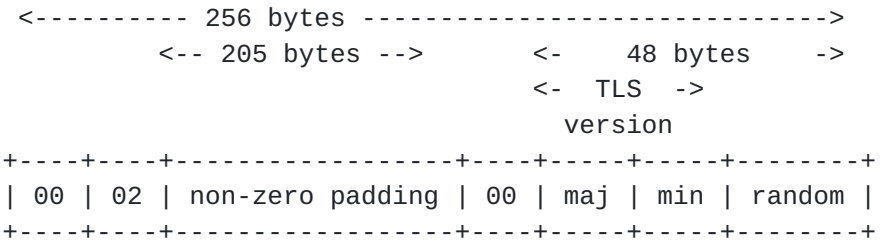
[RFC7627] defines the Extended Master Secret Extension where the "master_secret" is defined as follows:

```
 master_secret = PRF(pre_master_secret, "extended master secret",
                     session_hash)
                     [0..47];
 where:
    - session_hash = Hash(handshake_messages)
    - handshake_messages  is the concatenation of all the exchanged
      Handshake structures, as defined in Section 7.4 of [RFC5246].
    - Hash is as defined in Section 7.4.9 of [RFC5246]
```

As defined in section 8.1.1 [RFC2546], the pre_master is 48-byte generated by the TLS Client. The two first bytes indicates the TLS version and MUST be the same value as the one provided by the

ClientHello.client_version, and the remaining 46 bytes are expected to be random.

The pre_master is encrypted with the public key of the TLS Server as a EncryptedPreMasterSecret structure sent in the Client Key Exchange Message as described in section 7.4.7.1 [RFC5246]. The encryption follows for compatibility with previous TLS version RSAES-PKCS1-v1_5 scheme described in [RFC3447], which results in a 256 byte encrypted message for a 2048-bit RSA key or 128 byte encrypted message for a 1024 bit RSA key.



PKCS#1 padding for pre_master secret encrypted with 2048-bit RSA key

Upon receiving a Client Key Exchange Message with a KeyExchangeAlgorithm set to rsa, the Edge Server sends a request for the pre_master to the Key Server. The request provides the EncryptedPreMasterSecret as well as the ClientHello.client_version.

Upon receiving the EncryptedPreMasterSecret and the ClientHello.client_version, the Key Server decrypts the EncryptedPreMasterSecret following [RFC3447]. If the decryption is successful, the Key Server MUST check the version indicated in the two first bytes corresponds to the ClientHello.client_version as well as the length of the clear text pre_master. If one of the test fails, the Key Server MUST return an 'malformed request' error. If any other error occurs an 'unspecified error' MUST be returned. If it is successful, the Key Server returns the clear text of the pre_master.

Upon receiving the response or the error, the Edge Server proceeds as defined in [RFC2546]. If the pre_master is provided, the Edge Server computes the master_secret as defined in [RFC5246] or in [RFC7627]. If an error is returned, the Edge Server continue the exchange with a randomly generated pre_master.

DISCUSSION: if SKI is the interface between the Edge Server and the Key Server, maybe we could return the master_secret directly. Maybe an architecture with a Master Oracle and Key Server would better split the function between owning the private key - and only

decrypting - and providing the master with associate TLS syntax checking.

5.1.2. Ephemeral Diffie Hellman

[RFC5246] defines how the TLS Client and the Edge Server agrees for DHE_RSA. When the KeyExchangeAlgorythm has been agreed to dhe_rsa, as defined in section 7.4.3 of [RFC5246], the ServerKeyExchange message contains ServerDHParams as well as the Signature.

[RFC4492] defines the extension that enables the TLS Client and the Edge Server to agree ECDHE_RSA or ECDHE_ECDSA for the key exchange algorithm. When the KeyExchangeAlgorythm has been agreed to ec_diffie_hellman between the TLS Client and the Edge Server, as detailed in section 5.4 of [RFC4492], the ServerKeyExchange contains the ServerECDHParams and Signature.

In order to build the signature, the Edge Server provides Key Server the type of the key (ECHDE or DHE), the corresponding public key, the hash function, the signature algorithm to be used (RSA, or ECDSA), the ClientHello.random and the ServerHello.random.

Upon receiving the public key, the Key Server checks random numbers are 32bit long, and checks the validity of the public key. If the input data is not valid or has the wrong size, the Key Server MUST reply with a 'malformed request' error. Otherwise the Key Server hash and signs the output. If any error occurs during the signing process, the server responds with an 'unspecified error' error. If signing is successful, the server responds with the output data set to the result of the signing operation.

Upon receiving the response or the error, the Edge Server proceeds as defined in [RFC2546]. If the pre_master is provided, the Edge Server computes the master_secret as defined in [RFC5246] or in [RFC7627]. If an error is returned, the Edge Server continue the exchange with a randomly generated pre_master.

5.2. SKI Specification

The Session Key Interface is based on a request-response pattern where the Edge Server sends a SKI Request to the Key Server requesting a specific private key operation that the Edge Server needs to complete a TLS handshake. The Edge Server's request includes data to be processed, the identifier of the private key to be used, and any options necessary for the Key Server to correctly perform the requested operation. The Key Server answers with a SKI Response containing either the requested output data or an error.

Any request-response protocol can be used to carry the SKI payloads. Two obvious choices are the Hypertext Transfer Protocol (HTTP) [RFC7540] and the Constrained Application Protocol (CoAP) [RFC7252]. Which protocol to use is application specific. SKI requests are by default sent to the Request-URI '/ski'. The interface between the Edge Server and the Key Server MUST be protected by a security protocol providing integrity protection, confidentiality, and mutual authentication. If TLS is used, the implementation MUST fulfill at least the security requirements in [RFC7540] Section 9.2.

Two formats are defined for the SKI Payload format: the JavaScript Object Notation (JSON) [RFC7159] and the Concise Binary Object Representation (CBOR) [RFC7049]. In JSON, byte strings are Base64 encoded [RFC4648]. Which format to use is application specific. The payload consists of a single JSON or CBOR object consisting of one or more attribute-value pairs. The following attributes are defined:

'protocol' REQUIRED in SKI requests. Specifies the protocol version negotiated in the handshake between Client and Edge Server. Can take one of the values 'TLS 1.0', 'TLS 1.1', 'TLS 1.2', 'DTLS 1.0', or 'DTLS 1.2'.

'spki' REQUIRED in SKI requests. Byte string that identifies the Subject Public Key Info (SPKI) of a X.509 certificate [RFC5280] or a raw public key [RFC7250]. Contains a SHA-256 SPKI Fingerprint as defined in [RFC7469]

'method' Included in SKI requests to indicate the key exchange method. Can take one of the values 'ECDHE' or 'RSA'. MAY be omitted if the default value 'ECDHE' is used.

'hash' Included in SKI requests. MUST be used if a hash algorithm other than the default hash algorithm has been negotiated using the "signature_algorithms" extension. Can take one of the values 'SHA-224', 'SHA-256', 'SHA-384', or 'SHA-512'.

'input' REQUIRED in SKI requests. Byte string containing the input data to the private key operation. For static RSA it contains the encrypted premaster secret (EncryptedPreMasterSecret). For ECDHE it contains the data to be signed (ClientRandom + ServerRandom + ServerECDHParams).

'output' Included in successful SKI responses. Byte string containing the output data from the private key operation. For static RSA it contains the premaster secret (PreMasterSecret). For ECDHE is contains the signature (Signature).

'error' Included in SKI responses to indicate a fatal error. Can take one of the values 'request denied', 'spki not found', 'malformed request', or 'unspecified error'. SHALL not be sent together with 'output'.

5.2.1. Key Server Processing

The Key Server determines how to handle a SKI request based on the values provided for the 'protocol', 'spki', 'hash', and 'method' attributes. If the Key Server cannot parse the SKI request it MUST respond with a 'malformed request' error. If a private key matching the 'spki' value is not found, the Key Server MUST respond with a 'spki not found' error. If the Edge Server is not authorized to receive a response to the specific request, the Key Server MUST respond with a 'request denied' error.

DISCUSSION: For TLS1.0/DTLS1.0 only uses MD5 and SHA-1 are defined. SHA-256 only appears in TLS1.2. I suspect there are some additional checks to be done, or maybe that is fine to have TLS1.0 with these algorithms.

6. Interaction with TLS Extensions

Most TLS extensions interact seamlessly with SKI, but it is worth noting the few that do not:

[RFC6091] defines the use of OpenPGP certificates with TLS. As OpenPGP certificates do not have a SPKI field, SKI will not work with this extension unless the public key identification mechanism is updated.

[RFC6962] certificate transparency conflict with the proposed version of SKI since it requires signing of timestamps, while SKI only allows signing of valid ECDHE parameters.

A few other TLS extensions may have problems if a TLS client connects to different Edge Servers:

[RFC5077] defines session resumption with session tickets. As this extension uses a secret key stored on the server issuing the ticket, it only works if the resumption Edge Server has the same secret key.

[RFC5746] defines the renegotiation_info extension for secure renegotiation. As this extension is facilitated by binding the renegotiation to the previous connection, it only works if the renegotiation is done to the same Edge Server.

7. Examples

Note: Lengths of hexadecimal and base64 encoded strings in examples are not intended to be realistic. For readability, COSE objects are represented using CBOR's diagnostic notation [RFC7049].

7.1. ECDHE_ECDSA Key Exchange

If an ECDHE key exchange method is used, the Edge Server MUST receive the SKI Response before it can send the ServerKeyExchange message. An example message flow is shown in Figure 2.

Figure 2: Message Flow for ECDHE Key Exchange

7.1.1. SKI Request and Response with JSON/HTTP

SKI Request:

```
POST /ski HTTP/1.1
Host: keyserver.example.com
Content-Type: application/json
Content-Length: 166

{
  "protocol": "TLS 1.2",
  "method": "ECDHE",
  "hash": "SHA-256",
  "spki": "mPgHXSvrW6ygN4uhPnl0W2uGMSbCDjFV1bfkaVT5",
  "input": "Bn1eaonvIyCDFd9Ek8UyghL9SA1FXcDplnk8zNlLXBL4H0FAEFyvFO"
}
```

SKI Response:

```
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 62

{
  "output": "eysh5GCSbIjjHzDt7Co5PUuVnDePbUYI839yv30bJWquwJ3vyADor"
}
```

SKI Request:

```
POST /ski HTTP/1.1
Host: keyserver.example.com
Content-Type: application/json
Content-Length: 128

{
  "protocol": "TLS 1.1",
  "spki": "p8FU0McKWBBLEEFfQbnJPjW3Q6EcZ5t11cKKcuwj",
  "input": "yWCMO9P0yINtHUT17ZO1X1mUgwh1CrTGan9QaAGph9AnCO4HA44nez"
}
```

SKI Response:

```
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 62

{
  "output": "m7nJUltTVMiaQJyDcKPaq0ZOtfuRVnUt1cUx5KoP3w75MqpSelutO"
}
```

7.1.2. SKI Request and Response with CBOR/CoAP

SKI Request:

```
Header: POST (T=CON, Code=0.03, MID=0x1337)
Uri-Path: "ski"
Content-Format: 60 (application/cbor)
Payload: {
           "protocol": "TLS 1.0",
           "spki": h'a1fa7ec57a6a5485756c45ab58b2c992',
           "input": h'd2e61706059a16714e4716853e2917e34'
         }
```

SKI Response:

```
Header: 2.04 Changed (T=ACK, Code=2.04, MID=0x1337)
Content-Format: 60 (application/cbor)
Payload: { "output": h'2c8a0001b8295ab44d1930b8efdd9fb40' }
```

7.2. Static RSA Key Exchange

If the static RSA key exchange method is used, the Edge Server MUST receive the SKI Response before it can send the Finished message. An example message flow is shown in Figure 3.

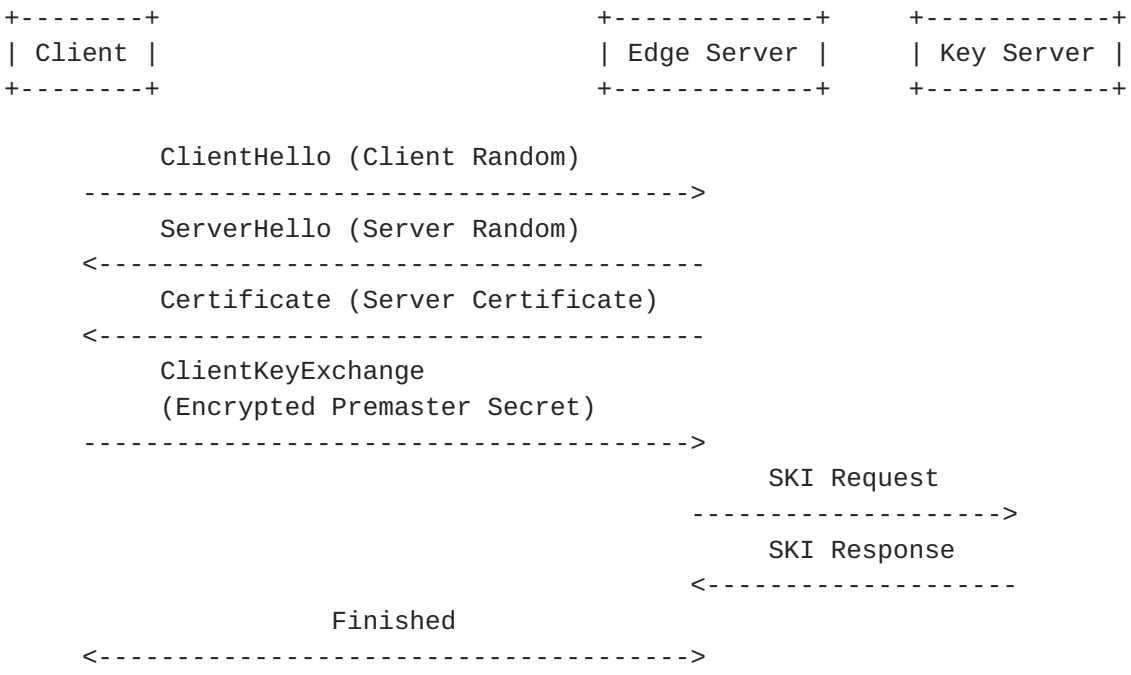


Figure 3: Message Flow for Static RSA Key Exchange

7.2.1. SKI Request and Response with JSON/HTTP

SKI Request:

```
POST /ski HTTP/1.1
Host: keyserver.example.com
Content-Type: application/json
Content-Length: 145

{
  "protocol": "TLS 1.2",
  "method": "RSA",
  "spki": "QItwmcEKcuMhCWIdESDPBbZtNgfwS7w84wizTk47",
  "input": "dEHffkdIoi2YhQmsqcum3kDk2cToQqO2JLzJVi4q8pJSvfSUyyhRv7"
}
```

SKI Response:

```
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 62

{
  "output": "CtehRGUae6NQ0daIuClSTg3nW62zqPvYTjnvIV0mt5kM49tIq9uDG"
}
```

7.2.2. SKI Request and Response with CBOR/CoAP

SKI Request:

```
Header: POST (T=CON, Code=0.03, MID=0xabba)
Uri-Path: "ski"
Content-Format: 60 (application/cbor)
Payload: {
           "protocol": "TLS 1.2",
           "method": "RSA",
           "spki": h'8378d0547da09484b8ae509565b0a595',
           "input": h'9da2d7a363ead429141f4dcad20befb6043'
         }
```

SKI Response:

```
Header: 2.04 Changed (T=ACK, Code=2.04, MID=0xabba)
Content-Format: 60 (application/cbor)
Payload: { "output" : h'827628ca533a1d1191acb0e106fb' }
```

8. IANA Considerations

This document defines the following. TODO...

9. Security Considerations

The security considerations in [RFC5246], [RFC4492], and [RFC7525] apply to this document as well.

The TLS Session Key Interface increases the security by making it possible to store private keys in a highly trusted location, physically separated from client facing servers. The main feature that separates TLS SKI from traditional TLS is the secure connection between the Edge Server and the Key Server. This connection is relied on to ensure that the servers are mutually authenticated and that the connection between them is private. A compromised Edge Server can still access client data as well as submit requests to the Key Server. However, the risks are reduces since no private keys can be compromised and the Key Server can at any time prevent the Edge Server from starting new TLS connections.

A compromised Edge Server could potentially launch timing side-channel attacks or buffer overflow attacks. And as the Key Server has limited knowledge of the input data it signs or decrypts, a compromised edge server could try to get the Key Server to process maliciously crafted input data resulting in a signed message or the decryption of the PreMasterSecret from another connection. However, these attacks are not introduced by SKI since they could be performed on a compromised traditional TLS server and, with the exception of the signing attack, can even be launched by a TLS client against an uncompromised TLS server.

10. Acknowledgements

The authors would like to thank Magnus Thulstrup and Hans Spaak for their valuable comments and feedback.

11. References

[HEART] Codenomicon, "The Heartbleed Bug", <http://heartbleed.com/>.

[RFC2246] Dierks, T. and C. Allen, "The TLS Protocol Version 1.0", RFC 2246, DOI 10.17487/RFC2246, January 1999, <http://www.rfc-editor.org/info/rfc2246>.

[RFC2546] Durand, A. and B. Buclin, "6Bone Routing Practice", RFC 2546, DOI 10.17487/RFC2546, March 1999, <http://www.rfc-editor.org/info/rfc2546>.

[RFC3447] Jonsson, J. and B. Kaliski, "Public-Key Cryptography Standards (PKCS) #1: RSA Cryptography Specifications Version 2.1", RFC 3447, DOI 10.17487/RFC3447, February 2003, <http://www.rfc-editor.org/info/rfc3447>.

[RFC4346] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.1", RFC 4346, DOI 10.17487/RFC4346, April 2006, <http://www.rfc-editor.org/info/rfc4346>.

[RFC4347] Rescorla, E. and N. Modadugu, "Datagram Transport Layer Security", RFC 4347, DOI 10.17487/RFC4347, April 2006, <http://www.rfc-editor.org/info/rfc4347>.

[RFC4492] Blake-Wilson, S., Bolyard, N., Gupta, V., Hawk, C., and B. Moeller, "Elliptic Curve Cryptography (ECC) Cipher Suites for Transport Layer Security (TLS)", RFC 4492, DOI 10.17487/RFC4492, May 2006, <http://www.rfc-editor.org/info/rfc4492>.

[RFC4648] Josefsson, S., "The Base16, Base32, and Base64 Data Encodings", RFC 4648, DOI 10.17487/RFC4648, October 2006, <http://www.rfc-editor.org/info/rfc4648>.

[RFC5054] Taylor, D., Wu, T., Mavrogiannopoulos, N., and T. Perrin, "Using the Secure Remote Password (SRP) Protocol for TLS Authentication", RFC 5054, DOI 10.17487/RFC5054, November 2007, <http://www.rfc-editor.org/info/rfc5054>.

[RFC5077] Salowey, J., Zhou, H., Eronen, P., and H. Tschofenig, "Transport Layer Security (TLS) Session Resumption without Server-Side State", RFC 5077, DOI 10.17487/RFC5077, January 2008, <http://www.rfc-editor.org/info/rfc5077>.

[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>.

[RFC5280] Cooper, D., Santesson, S., Farrell, S., Boeyen, S., Housley, R., and W. Polk, "Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile", RFC 5280, DOI 10.17487/RFC5280, May 2008, <http://www.rfc-editor.org/info/rfc5280>.

[RFC5746] Rescorla, E., Ray, M., Dispensa, S., and N. Oskov, "Transport Layer Security (TLS) Renegotiation Indication Extension", RFC 5746, DOI 10.17487/RFC5746, February 2010, <http://www.rfc-editor.org/info/rfc5746>.

[RFC6091] Mavrogiannopoulos, N. and D. Gillmor, "Using OpenPGP Keys for Transport Layer Security (TLS) Authentication", RFC 6091, DOI 10.17487/RFC6091, February 2011, <http://www.rfc-editor.org/info/rfc6091>.

[RFC6347] Rescorla, E. and N. Modadugu, "Datagram Transport Layer Security Version 1.2", RFC 6347, DOI 10.17487/RFC6347, January 2012, <http://www.rfc-editor.org/info/rfc6347>.

[RFC6962] Laurie, B., Langley, A., and E. Kasper, "Certificate Transparency", RFC 6962, DOI 10.17487/RFC6962, June 2013, <http://www.rfc-editor.org/info/rfc6962>.

[RFC7049] Bormann, C. and P. Hoffman, "Concise Binary Object Representation (CBOR)", RFC 7049, DOI 10.17487/RFC7049, October 2013, <http://www.rfc-editor.org/info/rfc7049>.

[RFC7159] Bray, T., Ed., "The JavaScript Object Notation (JSON) Data Interchange Format", RFC 7159, DOI 10.17487/RFC7159, March 2014, <http://www.rfc-editor.org/info/rfc7159>.

[RFC7250] Wouters, P., Ed., Tschofenig, H., Ed., Gilmore, J., Weiler, S., and T. Kivinen, "Using Raw Public Keys in Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS)", RFC 7250, DOI 10.17487/RFC7250, June 2014, <http://www.rfc-editor.org/info/rfc7250>.

[RFC7252] Shelby, Z., Hartke, K., and C. Bormann, "The Constrained Application Protocol (CoAP)", RFC 7252, DOI 10.17487/RFC7252, June 2014, <http://www.rfc-editor.org/info/rfc7252>.

[RFC7469] Evans, C., Palmer, C., and R. Sleevi, "Public Key Pinning Extension for HTTP", RFC 7469, DOI 10.17487/RFC7469, April 2015, <http://www.rfc-editor.org/info/rfc7469>.

[RFC7525] Sheffer, Y., Holz, R., and P. Saint-Andre, "Recommendations for Secure Use of Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS)", BCP 195, RFC 7525, DOI 10.17487/RFC7525, May 2015, <http://www.rfc-editor.org/info/rfc7525>.

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

[RFC7627] Bhargavan, K., Ed., Delignat-Lavaud, A., Pironti, A., Langley, A., and M. Ray, "Transport Layer Security (TLS) Session Hash and Extended Master Secret Extension", RFC 7627, DOI 10.17487/RFC7627, September 2015, <http://www.rfc-editor.org/info/rfc7627>.

Authors' Addresses

Kelsey Cairns
Washington State University
Pullman, WA 99164-2752
USA

Email: kcairns@wsu.edu

John Mattsson
Ericsson AB
SE-164 80 Stockholm
Sweden

Email: john.mattsson@ericsson.com

Robert Skog
Ericsson AB
SE-164 80 Stockholm
Sweden

Email: robert.skog@ericsson.com

Daniel Migault
Ericsson
8400 boulevard Decarie
Montreal, QC H4P 2N2
Canada

Phone: +1 514-452-2160
Email: daniel.migault@ericsson.com