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

Copyright Notice

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.

```
+------------+  Handshake  +-------------+    SKI    +------------+
| TLS Client | <---------> | Edge Server | <-------> | Key Server |
+------------+             +-------------+           +------------+
```

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.

```
 <---------- 256 bytes ------------------------------>
          <-- 205 bytes -->       <-    48 bytes    ->
                                  <-  TLS  ->
                                    version
+----+----+------------------+----+-----+-----+--------+
| 00 | 02 | non-zero padding | 00 | maj | min | random |
+----+----+------------------+----+-----+-----+--------+
```

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.

```
+--------+                             +-------------+     +------------+
| Client |                             | Edge Server |     | Key Server |
+--------+                             +-------------+     +------------+

        ClientHello (Client Random)
     ---------------------------------------->
        ServerHello (Server Random)
     <----------------------------------------
        Certificate (Server Certificate)
     <----------------------------------------
                                                   SKI Request
                                              --------------------->
                                                   SKI Response
        ServerKeyExchange                     <---------------------
        (ECDHParams, Signature)
     <----------------------------------------
        ClientKeyExchange (ClientDHPublic)
     ---------------------------------------->
        Finished
     <--------------------------------------->
```

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.

```
+--------+                            +-------------+     +------------+
| Client |                            | Edge Server |     | Key Server |
+--------+                            +-------------+     +------------+

          ClientHello (Client Random)
     --------------------------------------->
          ServerHello (Server Random)
     <---------------------------------------
          Certificate (Server Certificate)
     <---------------------------------------
          ClientKeyExchange
          (Encrypted Premaster Secret)
     --------------------------------------->
                                                   SKI Request
                                              -------------------->
                                                   SKI Response
                                              <--------------------
                     Finished
     <--------------------------------------->
```

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

TLS S. Santesson
Internet-Draft 3xA Security AB
Intended status: Standards Track H. Tschofenig
Expires: November 12, 2016 ARM Ltd.
May 11, 2016

Transport Layer Security (TLS) Cached Information Extension
draft-ietf-tls-cached-info-23.txt

Abstract

Transport Layer Security (TLS) handshakes often include fairly static information, such as the server certificate and a list of trusted certification authorities (CAs). This information can be of considerable size, particularly if the server certificate is bundled with a complete certificate chain (i.e., the certificates of intermediate CAs up to the root CA).

This document defines an extension that allows a TLS client to inform a server of cached information, allowing the server to omit already available information.

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

Reducing the amount of information exchanged during a Transport Layer Security handshake to a minimum helps to improve performance in environments where devices are connected to a network with a low bandwidth, and lossy radio technology. With Internet of Things such environments exist, for example, when devices use IEEE 802.15.4 or Bluetooth Smart. For more information about the challenges with smart object deployments please see [RFC6574].

This specification defines a TLS extension that allows a client and a server to exclude transmission information cached in an earlier TLS handshake.

A typical example exchange may therefore look as follows. First, the client and the server executes the full TLS handshake. The client then caches the certificate provided by the server. When the TLS client connects to the TLS server some time in the future, without using session resumption, it then attaches the cached_info extension defined in this document to the client hello message to indicate that it had cached the certificate, and it provides the fingerprint of it. If the server’s certificate has not changed then the TLS server does not need to send its certificate and the corresponding certificate chain again. In case information has changed, which can be seen from the fingerprint provided by the client, the certificate payload is transmitted to the client to allow the client to update the cache.

2. Terminology

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

This document refers to the TLS protocol but the description is equally applicable to DTLS as well.

3. Cached Information Extension

This document defines a new extension type (cached_info(TBD)), which is used in client hello and server hello messages. The extension type is specified as follows.

```
enum {
     cached_info(TBD), (65535)
} ExtensionType;
```

The extension_data field of this extension, when included in the client hello, MUST contain the CachedInformation structure. The client MAY send multiple CachedObjects of the same CachedInformationType. This may, for example, be the case when the client has cached multiple certificates from a server.

```
enum {
     cert(1), cert_req(2) (255)
} CachedInformationType;

struct {
     select (type) {
       case client:
         CachedInformationType type;
         opaque hash_value<1..255>;
       case server:
         CachedInformationType type;
     } body;
} CachedObject;

struct {
     CachedObject cached_info<1..2^16-1>;
} CachedInformation;
```

This document defines the following two types:

'cert' Type for not sending the complete Server Certificate Message:

With the type field set to 'cert', the client MUST include the fingerprint of the Certificate message in the hash_value field. For this type the fingerprint MUST be calculated using the procedure described in Section 5 with the Certificate message as input data.

'cert_req' Type for not sending the complete CertificateRequest Message:

With the type set to 'cert_req', the client MUST include the fingerprint of the CertificateRequest message in the hash_value field. For this type the fingerprint MUST be calculated using the procedure described in Section 5 with the CertificateRequest message as input data.

New cached info types can be added following the policy described in the IANA considerations section, see Section 8. New message digest algorithms for use with these types can also be added by registering a new type that makes use of the updated message digest algorithm. For practical reasons we recommend to re-use hash algorithms already available with TLS ciphersuites to avoid additional code and to keep the collision probably low new hash algorithms MUST NOT have a collision resistance worse than SHA-256.

4. Exchange Specification

Clients supporting this extension MAY include the "cached_info" extension in the (extended) client hello. If the client includes the extension then it MUST contain one or more CachedObject attributes.

A server supporting this extension MAY include the "cached_info" extension in the (extended) server hello. By returning the "cached_info" extension the server indicates that it supports the cached info types. For each indicated cached info type the server MUST alter the transmission of respective payloads, according to the rules outlined with each type. If the server includes the extension it MUST only include CachedObjects of a type also supported by the client (as expressed in the client hello). For example, if a client indicates support for 'cert' and 'cert_req' then the server cannot respond with a "cached_info" attribute containing support for ('foo-bar').

Since the client includes a fingerprint of information it cached (for each indicated type) the server is able to determine whether cached information is stale. If the server supports this specification and notices a mismatch between the data cached by the client and its own information then the server MUST include the information in full and MUST NOT list the respective type in the "cached_info" extension.

Note: If a server is part of a hosting environment then the client may have cached multiple data items for a single server. To allow the client to select the appropriate information from the cache it is RECOMMENDED that the client utilizes the Server Name Indication extension [RFC6066].

Following a successful exchange of the "cached_info" extension in the client and server hello, the server alters sending the corresponding handshake message. How information is altered from the handshake messages is defined in Section 4.1, and in Section 4.2 for the types defined in this specification.

Appendix A shows an example hash calculation and Section 6 shows an example protocol exchange.

4.1. Server Certificate Message

When a ClientHello message contains the "cached_info" extension with a type set to 'cert' then the server MAY send the Certificate message shown in Figure 1 under the following conditions:

- o The server software implements the "cached_info" extension defined in this specification.

- o The 'cert' cached info extension is enabled (for example, a policy allows the use of this extension).

- o The server compared the value in the hash_value field of the client-provided "cached_info" extension with the fingerprint of the Certificate message it normally sends to clients. This check ensures that the information cached by the client is current. The procedure for calculating the fingerprint is described in Section 5.

The original Certificate handshake message syntax is defined in [RFC5246] and has been extended with [RFC7250]. RFC 7250 allows the certificate payload to contain only the SubjectPublicKeyInfo instead of the full information typically found in a certificate. Hence, when this specification is used in combination with [RFC7250] and the negotiated certificate type is a raw public key then the TLS server omits sending a Certificate payload that contains an ASN.1 Certificate structure with the included SubjectPublicKeyInfo rather than the full certificate chain. As such, this extension is compatible with the raw public key extension defined in RFC 7250. Note: We assume that the server implementation is able to select the appropriate certificate or SubjectPublicKeyInfo from the received hash value. If the SNI extension is used by the client then the server has additional information to guide the selection of the appropriate cached info.

When the cached info specification is used then a modified version of the Certificate message is exchanged. The modified structure is shown in Figure 1.

```
struct {
    opaque hash_value<1..255>;
} Certificate;
```

Figure 1: Cached Info Certificate Message.

4.2. CertificateRequest Message

When a fingerprint for an object of type 'cert_req' is provided in the client hello, the server MAY send the CertificateRequest message shown in Figure 2 message under the following conditions:

- o The server software implements the "cached_info" extension defined in this specification.

- o The 'cert_req' cached info extension is enabled (for example, a policy allows the use of this extension).

- o The server compared the value in the hash_value field of the client-provided "cached_info" extension with the fingerprint of the CertificateRequest message it normally sends to clients. This check ensures that the information cached by the client is current. The procedure for calculating the fingerprint is described in Section 5.

- o The server wants to request a certificate from the client.

The original CertificateRequest handshake message syntax is defined in [RFC5246]. The modified structure of the CertificateRequest message is shown in Figure 2.

```
struct {
    opaque hash_value<1..255>;
} CertificateRequest;
```

Figure 2: Cached Info CertificateRequest Message.

The CertificateRequest payload is the input parameter to the fingerprint calculation described in Section 5.

5. Fingerprint Calculation

The fingerprint for the two cached info objects defined in this document MUST be computed as follows:

1. Compute the SHA-256 [RFC6234] hash of the input data. The input data depends on the cached info type. This document defines two cached info types, described in Section 4.1 and in Section 4.2. Note that the computed hash only covers the input data structure (and not any type and length information of the record layer). Appendix A shows an example.

2. Use the output of the SHA-256 hash.

The purpose of the fingerprint provided by the client is to help the server select the correct information. For example, in case of the certificate message the fingerprint identifies the server certificate (and the corresponding private key) for use for with the rest of the handshake. Servers may have more than one certificate and therefore a hash needs to be long enough to keep the probably of hash collisions low. On the other hand, the cached info design aims to reduce the amount of data being exchanged. The security of the handshake depends on the private key and not on the size of the fingerprint. Hence, the fingerprint is a way to prevent the server from accidentally selecting the wrong information. If an attacker injects an incorrect fingerprint then two outcomes are possible: (1) The fingerprint does not relate to any cached state and the server has to fall back to a full exchange. (2) If the attacker manages to inject a fingerprint that refers to data the client has not cached then the exchange will fail later when the client continues with the handshake and aims to verify the digital signature. The signature verification will fail since the public key cached by the client will not correspond to the private key that was used by server to sign the message.

6. Example

In the regular, full TLS handshake exchange, shown in Figure 3, the TLS server provides its certificate in the Certificate payload to the client, see step (1). This allows the client to store the certificate for future use. After some time the TLS client again interacts with the same TLS server and makes use of the TLS cached info extension, as shown in Figure 4. The TLS client indicates support for this specification via the "cached_info" extension, see step (2), and indicates that it has stored the certificate from the earlier exchange (by indicating the 'cert' type). With step (3) the TLS server acknowledges the supports of the 'cert' type and by including the value in the server hello informs the client that the content of the certificate payload contains the fingerprint of the certificate instead of the RFC 5246-defined payload of the certificate message in step (4).

```
ClientHello            ->
                       <-  ServerHello
                           Certificate* // (1)
                           ServerKeyExchange*
                           CertificateRequest*
                           ServerHelloDone

Certificate*
ClientKeyExchange
CertificateVerify*
[ChangeCipherSpec]
Finished               ->

                       <- [ChangeCipherSpec]
                          Finished

Application Data <-------> Application Data
```

Figure 3: Example Message Exchange: Initial (full) Exchange.

```
   ClientHello
   cached_info=(cert)     -> // (2)
                          <-  ServerHello
                              cached_info=(cert) (3)
                              Certificate (4)
                              ServerKeyExchange*
                              ServerHelloDone

   ClientKeyExchange
   CertificateVerify*
   [ChangeCipherSpec]
   Finished               ->

                          <- [ChangeCipherSpec]
                             Finished

   Application Data <-------> Application Data
```

Figure 4: Example Message Exchange: TLS Cached Extension Usage.

7. Security Considerations

This specification defines a mechanism to reference stored state using a fingerprint. Sending a fingerprint of cached information in an unencrypted handshake, as the client and server hello is, may allow an attacker or observer to correlate independent TLS exchanges. While some information elements used in this specification, such as server certificates, are public objects and usually do not contain sensitive information, other not yet defined types may. Those who implement and deploy this specification should therefore make an informed decision whether the cached information is inline with their security and privacy goals. In case of concerns, it is advised to avoid sending the fingerprint of the data objects in clear.

The use of the cached info extension allows the server to send significantly smaller TLS messages. Consequently, these omitted parts of the messages are not included in the transcript of the handshake in the TLS Finish message. However, since the client and the server communicate the hash values of the cached data in the initial handshake messages the fingerprints are included in the TLS Finish message.

Clients MUST ensure that they only cache information from legitimate sources. For example, when the client populates the cache from a TLS exchange then it must only cache information after the successful completion of a TLS exchange to ensure that an attacker does not inject incorrect information into the cache. Failure to do so allows for man-in-the-middle attacks.

Security considerations for the fingerprint calculation are discussed in Section 5.

8. IANA Considerations

8.1. New Entry to the TLS ExtensionType Registry

IANA is requested to add an entry to the existing TLS ExtensionType registry, defined in [RFC5246], for cached_info(TBD) defined in this document.

8.2. New Registry for CachedInformationType

IANA is requested to establish a registry for TLS CachedInformationType values. The first entries in the registry are

- cert(1)

- cert_req(2)

The policy for adding new values to this registry, following the terminology defined in [RFC5226], is as follows:

- 0-63 (decimal): Standards Action

- 64-223 (decimal): Specification Required

- 224-255 (decimal): reserved for Private Use

9. Acknowledgments

We would like to thank the following persons for your detailed document reviews:

- Paul Wouters and Nikos Mavrogiannopoulos (December 2011)

- Rob Stradling (February 2012)

- Ondrej Mikle (March 2012)

- Ilari Liusvaara, Adam Langley, and Eric Rescorla (July 2014)

- Sean Turner (August 2014)

- Martin Thomson (August 2015)

- Jouni Korhonen (November 2015)

o Matt Miller (December 2015)

We would also to thank Martin Thomson, Karthikeyan Bhargavan, Sankalp Bagaria and Eric Rescorla for their feedback regarding the fingerprint calculation.

Finally, we would like to thank the TLS working group chairs, Sean Turner and Joe Salowey, as well as the responsible security area director, Stephen Farrell, for their support and their reviews.

10. References

10.1. Normative References

[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>.

[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>.

[RFC6066] Eastlake 3rd, D., "Transport Layer Security (TLS) Extensions: Extension Definitions", RFC 6066, DOI 10.17487/RFC6066, January 2011, <http://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, <http://www.rfc-editor.org/info/rfc6234>.

10.2. Informative References

[ASN.1-Dump]
Gutmann, P., "ASN.1 Object Dump Program", February 2013, <http://www.cs.auckland.ac.nz/~pgut001/>.

[RFC5226] Narten, T. and H. Alvestrand, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 5226, DOI 10.17487/RFC5226, May 2008, <http://www.rfc-editor.org/info/rfc5226>.

[RFC6574] Tschofenig, H. and J. Arkko, "Report from the Smart Object Workshop", RFC 6574, DOI 10.17487/RFC6574, April 2012, <http://www.rfc-editor.org/info/rfc6574>.

[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>.

Appendix A. Example

Consider a certificate containing an NIST P256 elliptic curve public key displayed using Peter Gutmann's ASN.1 decoder [ASN.1-Dump] in Figure 5.

```
  0 556: SEQUENCE {
  4 434:   SEQUENCE {
  8   3:     [0] {
 10   1:       INTEGER 2
       :       }
 13   1:     INTEGER 13
 16  10:     SEQUENCE {
 18   8:      OBJECT IDENTIFIER ecdsaWithSHA256 (1 2 840 10045 4 3 2)
       :       }
 28  62:     SEQUENCE {
 30  11:       SET {
 32   9:         SEQUENCE {
 34   3:           OBJECT IDENTIFIER countryName (2 5 4 6)
 39   2:           PrintableString 'NL'
       :           }
       :         }
 43  17:       SET {
 45  15:         SEQUENCE {
 47   3:           OBJECT IDENTIFIER organizationName (2 5 4 10)
 52   8:           PrintableString 'PolarSSL'
       :           }
       :         }
 62  28:       SET {
 64  26:         SEQUENCE {
 66   3:           OBJECT IDENTIFIER commonName (2 5 4 3)
 71  19:           PrintableString 'Polarssl Test EC CA'
       :           }
       :         }
       :       }
 92  30:     SEQUENCE {
 94  13:       UTCTime 24/09/2013 15:52:04 GMT
109  13:       UTCTime 22/09/2023 15:52:04 GMT
       :       }
124  65:     SEQUENCE {
126  11:       SET {
```

```
  128   9:         SEQUENCE {
  130   3:           OBJECT IDENTIFIER countryName (2 5 4 6)
  135   2:           PrintableString 'NL'
         :           }
         :         }
  139  17:       SET {
  141  15:         SEQUENCE {
  143   3:           OBJECT IDENTIFIER organizationName (2 5 4 10)
  148   8:           PrintableString 'PolarSSL'
         :           }
         :         }
  158  31:       SET {
  160  29:         SEQUENCE {
  162   3:           OBJECT IDENTIFIER commonName (2 5 4 3)
  167  22:           PrintableString 'PolarSSL Test Client 2'
         :           }
         :         }
         :       }
  191  89:     SEQUENCE {
  193  19:       SEQUENCE {
  195   7:         OBJECT IDENTIFIER ecPublicKey (1 2 840 10045 2 1)
  204   8:         OBJECT IDENTIFIER prime256v1 (1 2 840 10045 3 1 7)
         :         }
  214  66:       BIT STRING
         :         04 57 E5 AE B1 73 DF D3 AC BB 93 B8 81 FF 12 AE
         :         EE E6 53 AC CE 55 53 F6 34 0E CC 2E E3 63 25 0B
         :         DF 98 E2 F3 5C 60 36 96 C0 D5 18 14 70 E5 7F 9F
         :         D5 4B 45 18 E5 B0 6C D5 5C F8 96 8F 87 70 A3 E4
         :         C7
         :       }
  282 157:     [3] {
  285 154:       SEQUENCE {
  288   9:         SEQUENCE {
  290   3:           OBJECT IDENTIFIER basicConstraints (2 5 29 19)
  295   2:           OCTET STRING, encapsulates {
  297   0:             SEQUENCE {}
         :             }
         :           }
  299  29:         SEQUENCE {
  301   3:           OBJECT IDENTIFIER subjectKeyIdentifier (2 5 29 14)
  306  22:           OCTET STRING, encapsulates {
  308  20:             OCTET STRING
         :              7A 00 5F 86 64 FC E0 5D E5 11 10 3B B2 E6 3B C4
         :              26 3F CF E2
         :             }
         :           }
  330 110:         SEQUENCE {
  332   3:          OBJECT IDENTIFIER authorityKeyIdentifier (2 5 29 35)
```

```
  337 103:          OCTET STRING, encapsulates {
  339 101:             SEQUENCE {
  341  20:               [0]
         :               9D 6D 20 24 49 01 3F 2B CB 78 B5 19 BC 7E 24
         :               C9 DB FB 36 7C
  363  66:               [1] {
  365  64:                 [4] {
  367  62:                   SEQUENCE {
  369  11:                     SET {
  371   9:                      SEQUENCE {
  373   3:                       OBJECT IDENTIFIER countryName (2 5 4 6)
  378   2:                       PrintableString 'NL'
         :                        }
         :                      }
  382  17:                     SET {
  384  15:                      SEQUENCE {
  386   3:                        OBJECT IDENTIFIER organizationName
         :                               (2 5 4 10)
  391   8:                        PrintableString 'PolarSSL'
         :                        }
         :                      }
  401  28:                     SET {
  403  26:                      SEQUENCE {
  405   3:                       OBJECT IDENTIFIER commonName (2 5 4 3)
  410  19:                       PrintableString 'Polarssl Test EC CA'
         :                        }
         :                      }
         :                     }
         :                   }
         :                 }
  431   9:               [2] 00 C1 43 E2 7E 62 43 CC E8
         :               }
         :             }
         :           }
         :         }
         :       }
         :     }
  442  10:   SEQUENCE {
  444   8:     OBJECT IDENTIFIER ecdsaWithSHA256 (1 2 840 10045 4 3 2)
         :     }
  454 104:   BIT STRING, encapsulates {
  457 101:     SEQUENCE {
  459  48:       INTEGER
         :         4A 65 0D 7B 20 83 A2 99 B9 A8 0F FC 8D EE 8F 3D
         :         BB 70 4C 96 03 AC 8E 78 70 DD F2 0E A0 B2 16 CB
         :         65 8E 1A C9 3F 2C 61 7E F8 3C EF AD 1C EE 36 20
  509  49:       INTEGER
         :         00 9D F2 27 A6 D5 74 B8 24 AE E1 6A 3F 31 A1 CA
```

```
         :         54 2F 08 D0 8D EE 4F 0C 61 DF 77 78 7D B4 FD FC
         :         42 49 EE E5 B2 6A C2 CD 26 77 62 8E 28 7C 9E 57
         :         45
         :       }
         :     }
         :   }
```

Figure 5: ASN.1-based Certificate: Example.

To include the certificate shown in Figure 5 in a TLS/DTLS Certificate message it is prepended with a message header. This Certificate message header in our example is 0b 00 02 36 00 02 33 00 02 00 02 30, which indicates:

Message Type: 0b -- 1 byte type field indicating a Certificate message

Length: 00 02 36 -- 3 byte length field indicating a 566 bytes payload

Certificates Length: 00 02 33 -- 3 byte length field indicating 563 bytes for the entire certificates_list structure, which may contain multiple certificates. In our example only one certificate is included.

Certificate Length: 00 02 30 -- 3 byte length field indicating 560 bytes of the actual certificate following immediately afterwards. In our example, this is the certificate content with 30 82 02 .... 9E 57 45 shown in Figure 6.

The hex encoding of the ASN.1 encoded certificate payload shown in Figure 5 leads to the following encoding.

```
30 82 02 2C 30 82 01 B2  A0 03 02 01 02 02 01 0D
30 0A 06 08 2A 86 48 CE  3D 04 03 02 30 3E 31 0B
30 09 06 03 55 04 06 13  02 4E 4C 31 11 30 0F 06
03 55 04 0A 13 08 50 6F  6C 61 72 53 53 4C 31 1C
30 1A 06 03 55 04 03 13  13 50 6F 6C 61 72 73 73
6C 20 54 65 73 74 20 45  43 20 43 41 30 1E 17 0D
31 33 30 39 32 34 31 35  35 32 30 34 5A 17 0D 32
33 30 39 32 32 31 35 35  32 30 34 5A 30 41 31 0B
30 09 06 03 55 04 06 13  02 4E 4C 31 11 30 0F 06
03 55 04 0A 13 08 50 6F  6C 61 72 53 53 4C 31 1F
30 1D 06 03 55 04 03 13  16 50 6F 6C 61 72 53 53
4C 20 54 65 73 74 20 43  6C 69 65 6E 74 20 32 30
59 30 13 06 07 2A 86 48  CE 3D 02 01 06 08 2A 86
48 CE 3D 03 01 07 03 42  00 04 57 E5 AE B1 73 DF
D3 AC BB 93 B8 81 FF 12  AE EE E6 53 AC CE 55 53
F6 34 0E CC 2E E3 63 25  0B DF 98 E2 F3 5C 60 36
96 C0 D5 18 14 70 E5 7F  9F D5 4B 45 18 E5 B0 6C
D5 5C F8 96 8F 87 70 A3  E4 C7 A3 81 9D 30 81 9A
30 09 06 03 55 1D 13 04  02 30 00 30 1D 06 03 55
1D 0E 04 16 04 14 7A 00  5F 86 64 FC E0 5D E5 11
10 3B B2 E6 3B C4 26 3F  CF E2 30 6E 06 03 55 1D
23 04 67 30 65 80 14 9D  6D 20 24 49 01 3F 2B CB
78 B5 19 BC 7E 24 C9 DB  FB 36 7C A1 42 A4 40 30
3E 31 0B 30 09 06 03 55  04 06 13 02 4E 4C 31 11
30 0F 06 03 55 04 0A 13  08 50 6F 6C 61 72 53 53
4C 31 1C 30 1A 06 03 55  04 03 13 13 50 6F 6C 61
72 73 73 6C 20 54 65 73  74 20 45 43 20 43 41 82
09 00 C1 43 E2 7E 62 43  CC E8 30 0A 06 08 2A 86
48 CE 3D 04 03 02 03 68  00 30 65 02 30 4A 65 0D
7B 20 83 A2 99 B9 A8 0F  FC 8D EE 8F 3D BB 70 4C
96 03 AC 8E 78 70 DD F2  0E A0 B2 16 CB 65 8E 1A
C9 3F 2C 61 7E F8 3C EF  AD 1C EE 36 20 02 31 00
9D F2 27 A6 D5 74 B8 24  AE E1 6A 3F 31 A1 CA 54
2F 08 D0 8D EE 4F 0C 61  DF 77 78 7D B4 FD FC 42
49 EE E5 B2 6A C2 CD 26  77 62 8E 28 7C 9E 57 45
```

Figure 6: Hex Encoding of the Example Certificate.

Applying the SHA-256 hash function to the Certificate message, which is starts with 0b 00 02 and ends with 9E 57 45, produces 0x086eefb4859adfe977defac494fff6b73033b4ce1f86b8f2a9fc0c6bf98605af.

Authors' Addresses

Stefan Santesson
3xA Security AB
Scheelev. 17
Lund 223 70
Sweden

Email: sts@aaa-sec.com

Hannes Tschofenig
ARM Ltd.
Hall in Tirol 6060
Austria

Email: Hannes.tschofenig@gmx.net
URI: http://www.tschofenig.priv.at

TLS Working Group
Internet-Draft
Obsoletes: 4492 (if approved)
Intended status: Standards Track
Expires: November 6, 2017

Y. Nir
Check Point
S. Josefsson
SJD AB
M. Pegourie-Gonnard
Independent / PolarSSL

May 5, 2017

Elliptic Curve Cryptography (ECC) Cipher Suites for Transport Layer Security (TLS) Versions 1.2 and Earlier
draft-ietf-tls-rfc4492bis-17

Abstract

This document describes key exchange algorithms based on Elliptic Curve Cryptography (ECC) for the Transport Layer Security (TLS) protocol. In particular, it specifies the use of Ephemeral Elliptic Curve Diffie-Hellman (ECDHE) key agreement in a TLS handshake and the use of Elliptic Curve Digital Signature Algorithm (ECDSA) and Edwards Digital Signature Algorithm (EdDSA) as authentication mechanisms.

This document obsoletes and replaces RFC 4492.

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

This document describes additions to TLS to support ECC, applicable to TLS versions 1.0 [RFC2246], 1.1 [RFC4346], and 1.2 [RFC5246]. The use of ECC in TLS 1.3 is defined in [I-D.ietf-tls-tls13], and is explicitly out of scope for this document. In particular, this document defines:

- o the use of the ECDHE key agreement scheme with ephemeral keys to establish the TLS premaster secret, and
- o the use of ECDSA and EdDSA signatures for authentication of TLS peers.

The remainder of this document is organized as follows. Section 2 provides an overview of ECC-based key exchange algorithms for TLS. Section 3 describes the use of ECC certificates for client authentication. TLS extensions that allow a client to negotiate the use of specific curves and point formats are presented in Section 4. Section 5 specifies various data structures needed for an ECC-based handshake, their encoding in TLS messages, and the processing of those messages. Section 6 defines ECC-based cipher suites and identifies a small subset of these as recommended for all implementations of this specification. Section 8 discusses security considerations. Section 9 describes IANA considerations for the name spaces created by this document’s predecessor. Section 10 gives acknowledgements. Appendix B provides differences from [RFC4492], the document that this one replaces.

Implementation of this specification requires familiarity with TLS, TLS extensions [RFC4366], and ECC.

1.1. Conventions Used in This Document

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. Key Exchange Algorithm

This document defines three new ECC-based key exchange algorithms for TLS. All of them use Ephemeral ECDH (ECDHE) to compute the TLS premaster secret, and they differ only in the mechanism (if any) used to authenticate them. The derivation of the TLS master secret from the premaster secret and the subsequent generation of bulk encryption/MAC keys and initialization vectors is independent of the key exchange algorithm and not impacted by the introduction of ECC.

Table 1 summarizes the new key exchange algorithms. All of these key exchange algorithms provide forward secrecy if and only if fresh ephemeral keys are generated and used, and also destroyed after use.

Algorithm	Description
ECDHE_ECDSA	Ephemeral ECDH with ECDSA or EdDSA signatures.
ECDHE_RSA	Ephemeral ECDH with RSA signatures.
ECDH_anon	Anonymous ephemeral ECDH, no signatures.

Table 1: ECC Key Exchange Algorithms

These key exchanges are analogous to DHE_DSS, DHE_RSA, and DH_anon, respectively.

With ECDHE_RSA, a server can reuse its existing RSA certificate and easily comply with a constrained client's elliptic curve preferences (see Section 4). However, the computational cost incurred by a server is higher for ECDHE_RSA than for the traditional RSA key exchange, which does not provide forward secrecy.

The anonymous key exchange algorithm does not provide authentication of the server or the client. Like other anonymous TLS key exchanges, it is subject to man-in-the-middle attacks. Applications using TLS with this algorithm SHOULD provide authentication by other means.

```
Client                                        Server
------                                        ------
ClientHello          -------->
                                         ServerHello
                                        Certificate*
                                  ServerKeyExchange*
                                CertificateRequest*+
                     <--------       ServerHelloDone
Certificate*+
ClientKeyExchange
CertificateVerify*+
[ChangeCipherSpec]
Finished             -------->
                                  [ChangeCipherSpec]
                     <--------              Finished
Application Data     <------->      Application Data
     * message is not sent under some conditions
     + message is not sent unless client authentication
       is desired
```

Figure 1: Message flow in a full TLS 1.2 handshake

Figure 1 shows all messages involved in the TLS key establishment protocol (aka full handshake). The addition of ECC has direct impact only on the ClientHello, the ServerHello, the server's Certificate message, the ServerKeyExchange, the ClientKeyExchange, the CertificateRequest, the client's Certificate message, and the CertificateVerify. Next, we describe the ECC key exchange algorithm in greater detail in terms of the content and processing of these messages. For ease of exposition, we defer discussion of client authentication and associated messages (identified with a + in Figure 1) until Section 3 and of the optional ECC-specific extensions (which impact the Hello messages) until Section 4.

2.1. ECDHE_ECDSA

In ECDHE_ECDSA, the server's certificate MUST contain an ECDSA- or EdDSA-capable public key.

The server sends its ephemeral ECDH public key and a specification of the corresponding curve in the ServerKeyExchange message. These parameters MUST be signed with ECDSA or EdDSA using the private key corresponding to the public key in the server's Certificate.

The client generates an ECDH key pair on the same curve as the server's ephemeral ECDH key and sends its public key in the ClientKeyExchange message.

Both client and server perform an ECDH operation Section 5.10 and use the resultant shared secret as the premaster secret.

2.2. ECDHE_RSA

This key exchange algorithm is the same as ECDHE_ECDSA except that the server's certificate MUST contain an RSA public key authorized for signing, and that the signature in the ServerKeyExchange message must be computed with the corresponding RSA private key.

2.3. ECDH_anon

NOTE: Despite the name beginning with "ECDH_" (no E), the key used in ECDH_anon is ephemeral just like the key in ECDHE_RSA and ECDHE_ECDSA. The naming follows the example of DH_anon, where the key is also ephemeral but the name does not reflect it.

In ECDH_anon, the server's Certificate, the CertificateRequest, the client's Certificate, and the CertificateVerify messages MUST NOT be sent.

The server MUST send an ephemeral ECDH public key and a specification of the corresponding curve in the ServerKeyExchange message. These parameters MUST NOT be signed.

The client generates an ECDH key pair on the same curve as the server's ephemeral ECDH key and sends its public key in the ClientKeyExchange message.

Both client and server perform an ECDH operation and use the resultant shared secret as the premaster secret. All ECDH calculations are performed as specified in Section 5.10.

2.4. Algorithms in Certificate Chains

This specification does not impose restrictions on signature schemes used anywhere in the certificate chain. The previous version of this document required the signatures to match, but this restriction, originating in previous TLS versions is lifted here as it had been in RFC 5246.

3. Client Authentication

This document defines a client authentication mechanism, named after the type of client certificate involved: ECDSA_sign. The ECDSA_sign mechanism is usable with any of the non-anonymous ECC key exchange algorithms described in Section 2 as well as other non-anonymous (non-ECC) key exchange algorithms defined in TLS.

Note that client certificates with EdDSA public keys also use this mechanism.

The server can request ECC-based client authentication by including this certificate type in its CertificateRequest message. The client must check if it possesses a certificate appropriate for the method suggested by the server and is willing to use it for authentication.

If these conditions are not met, the client SHOULD send a client Certificate message containing no certificates. In this case, the ClientKeyExchange MUST be sent as described in Section 2, and the CertificateVerify MUST NOT be sent. If the server requires client authentication, it may respond with a fatal handshake failure alert.

If the client has an appropriate certificate and is willing to use it for authentication, it must send that certificate in the client's Certificate message (as per Section 5.6) and prove possession of the private key corresponding to the certified key. The process of determining an appropriate certificate and proving possession is different for each authentication mechanism and described below.

NOTE: It is permissible for a server to request (and the client to send) a client certificate of a different type than the server certificate.

3.1. ECDSA_sign

To use this authentication mechanism, the client MUST possess a certificate containing an ECDSA- or EdDSA-capable public key.

The client proves possession of the private key corresponding to the certified key by including a signature in the CertificateVerify message as described in Section 5.8.

4. TLS Extensions for ECC

Two TLS extensions are defined in this specification: (i) the Supported Elliptic Curves Extension, and (ii) the Supported Point Formats Extension. These allow negotiating the use of specific curves and point formats (e.g., compressed vs. uncompressed, respectively) during a handshake starting a new session. These extensions are especially relevant for constrained clients that may only support a limited number of curves or point formats. They follow the general approach outlined in [RFC4366]; message details are specified in Section 5. The client enumerates the curves it supports and the point formats it can parse by including the appropriate extensions in its ClientHello message. The server similarly enumerates the point formats it can parse by including an extension in its ServerHello message.

A TLS client that proposes ECC cipher suites in its ClientHello message SHOULD include these extensions. Servers implementing ECC cipher suites MUST support these extensions, and when a client uses these extensions, servers MUST NOT negotiate the use of an ECC cipher suite unless they can complete the handshake while respecting the choice of curves specified by the client. This eliminates the possibility that a negotiated ECC handshake will be subsequently aborted due to a client's inability to deal with the server's EC key.

The client MUST NOT include these extensions in the ClientHello message if it does not propose any ECC cipher suites. A client that proposes ECC cipher suites may choose not to include these extensions. In this case, the server is free to choose any one of the elliptic curves or point formats listed in Section 5. That section also describes the structure and processing of these extensions in greater detail.

In the case of session resumption, the server simply ignores the Supported Elliptic Curves Extension and the Supported Point Formats Extension appearing in the current ClientHello message. These extensions only play a role during handshakes negotiating a new session.

5. Data Structures and Computations

This section specifies the data structures and computations used by ECC-based key mechanisms specified in the previous three sections. The presentation language used here is the same as that used in TLS. Since this specification extends TLS, these descriptions should be merged with those in the TLS specification and any others that extend TLS. This means that enum types may not specify all possible values, and structures with multiple formats chosen with a select() clause may not indicate all possible cases.

5.1. Client Hello Extensions

This section specifies two TLS extensions that can be included with the ClientHello message as described in [RFC4366], the Supported Elliptic Curves Extension and the Supported Point Formats Extension.

When these extensions are sent:

The extensions SHOULD be sent along with any ClientHello message that proposes ECC cipher suites.

Meaning of these extensions:

These extensions allow a client to enumerate the elliptic curves it supports and/or the point formats it can parse.

Structure of these extensions:

The general structure of TLS extensions is described in [RFC4366], and this specification adds two types to ExtensionType.

```
enum {
    elliptic_curves(10),
    ec_point_formats(11)
} ExtensionType;
```

- elliptic_curves (Supported Elliptic Curves Extension): Indicates the set of elliptic curves supported by the client. For this extension, the opaque extension_data field contains NamedCurveList. See Section 5.1.1 for details.
- ec_point_formats (Supported Point Formats Extension): Indicates the set of point formats that the client can parse. For this extension, the opaque extension_data field contains ECPointFormatList. See Section 5.1.2 for details.

Actions of the sender:

A client that proposes ECC cipher suites in its ClientHello message appends these extensions (along with any others), enumerating the curves it supports and the point formats it can parse. Clients SHOULD send both the Supported Elliptic Curves Extension and the Supported Point Formats Extension. If the Supported Point Formats Extension is indeed sent, it MUST contain the value 0 (uncompressed) as one of the items in the list of point formats.

Actions of the receiver:

A server that receives a ClientHello containing one or both of these extensions MUST use the client's enumerated capabilities to guide its selection of an appropriate cipher suite. One of the proposed ECC cipher suites must be negotiated only if the server can successfully complete the handshake while using the curves and point formats supported by the client (cf. Section 5.3 and Section 5.4).

NOTE: A server participating in an ECDHE_ECDSA key exchange may use different curves for the ECDSA or EdDSA key in its certificate, and for the ephemeral ECDH key in the ServerKeyExchange message. The server MUST consider the extensions in both cases.

If a server does not understand the Supported Elliptic Curves Extension, does not understand the Supported Point Formats Extension, or is unable to complete the ECC handshake while restricting itself to the enumerated curves and point formats, it MUST NOT negotiate the use of an ECC cipher suite. Depending on what other cipher suites are proposed by the client and supported by the server, this may result in a fatal handshake failure alert due to the lack of common cipher suites.

5.1.1. Supported Elliptic Curves Extension

RFC 4492 defined 25 different curves in the NamedCurve registry (now renamed the "Supported Groups" registry, although the enumeration below is still named NamedCurve) for use in TLS. Only three have seen much use. This specification is deprecating the rest (with numbers 1-22). This specification also deprecates the explicit curves with identifiers 0xFF01 and 0xFF02. It also adds the new curves defined in [RFC7748]. The end result is as follows:

```
enum {
    deprecated(1..22),
    secp256r1 (23), secp384r1 (24), secp521r1 (25),
    x25519(29), x448(30),
    reserved (0xFE00..0xFEFF),
    deprecated(0xFF01..0xFF02),
    (0xFFFF)
} NamedCurve;
```

Note that other specifications have since added other values to this enumeration. Some of those values are not curves at all, but finite field groups. See [RFC7919].

secp256r1, etc: Indicates support of the corresponding named curve or groups. The named curves secp256r1, secp384r1, and secp521r1 are specified in SEC 2 [SECG-SEC2]. These curves are also recommended in ANSI X9.62 [ANSI.X9-62.2005] and FIPS 186-4 [FIPS.186-4]. The rest of this document refers to these three curves as the "NIST curves" because they were originally standardized by the National Institute of Standards and Technology. The curves x25519 and x448 are defined in [RFC7748]. Values 0xFE00 through 0xFEFF are reserved for private use.

The predecessor of this document also supported explicitly defined prime and char2 curves, but these are deprecated by this specification.

The NamedCurve name space is maintained by IANA. See Section 9 for information on how new value assignments are added.

```
struct {
    NamedCurve named_curve_list<2..2^16-1>
} NamedCurveList;
```

Items in named_curve_list are ordered according to the client's preferences (favorite choice first).

As an example, a client that only supports secp256r1 (aka NIST P-256; value 23 = 0x0017) and secp384r1 (aka NIST P-384; value 24 = 0x0018) and prefers to use secp256r1 would include a TLS extension consisting of the following octets. Note that the first two octets indicate the extension type (Supported Elliptic Curves Extension):

```
00 0A 00 06 00 04 00 17 00 18
```

5.1.2. Supported Point Formats Extension

```
enum {
    uncompressed (0),
    deprecated (1..2),
    reserved (248..255)
} ECPointFormat;
struct {
    ECPointFormat ec_point_format_list<1..2^8-1>
} ECPointFormatList;
```

Three point formats were included in the definition of ECPointFormat above. This specification deprecates all but the uncompressed point format. Implementations of this document MUST support the uncompressed format for all of their supported curves, and MUST NOT support other formats for curves defined in this specification. For backwards compatibility purposes, the point format list extension MAY still be included, and contain exactly one value: the uncompressed point format (0). RFC 4492 specified that if this extension is missing, it means that only the uncompressed point format is supported, so interoperability with implementations that support the uncompressed format should work with or without the extension.

If the client sends the extension and the extension does not contain the uncompressed point format, and the client has used the Supported Groups extension to indicate support for any of the curves defined in this specification then the server MUST abort the handshake and return an illegal_parameter alert.

The ECPointFormat name space is maintained by IANA. See Section 9 for information on how new value assignments are added.

A client compliant with this specification that supports no other curves MUST send the following octets; note that the first two octets indicate the extension type (Supported Point Formats Extension):

```
00 0B 00 02 01 00
```

5.1.3. The signature_algorithms Extension and EdDSA

The signature_algorithms extension, defined in section 7.4.1.4.1 of [RFC5246], advertises the combinations of signature algorithm and hash function that the client supports. The pure (non pre-hashed) forms of EdDSA do not hash the data before signing it. For this reason it does not make sense to combine them with a signature algorithm in the extension.

For bits-on-the-wire compatibility with TLS 1.3, we define a new dummy value in the HashAlgorithm registry which we will call "Intrinsic" (value TBD5) meaning that hashing is intrinsic to the signature algorithm.

To represent ed25519 and ed448 in the signature_algorithms extension, the value shall be (TBD5,TBD3) and (TBD5,TBD4) respectively.

5.2. Server Hello Extension

This section specifies a TLS extension that can be included with the ServerHello message as described in [RFC4366], the Supported Point Formats Extension.

When this extension is sent:

The Supported Point Formats Extension is included in a ServerHello message in response to a ClientHello message containing the Supported Point Formats Extension when negotiating an ECC cipher suite.

Meaning of this extension:

This extension allows a server to enumerate the point formats it can parse (for the curve that will appear in its ServerKeyExchange message when using the ECDHE_ECDSA, ECDHE_RSA, or ECDH_anon key exchange algorithm.

Structure of this extension:

The server’s Supported Point Formats Extension has the same structure as the client’s Supported Point Formats Extension (see Section 5.1.2). Items in ec_point_format_list here are ordered according to the server’s preference (favorite choice first). Note that the server MAY include items that were not found in the client's list. However, without extensions this specification allows exactly one point format, so there is not really any opportunity for mismatches.

Actions of the sender:

A server that selects an ECC cipher suite in response to a ClientHello message including a Supported Point Formats Extension appends this extension (along with others) to its ServerHello message, enumerating the point formats it can parse. The Supported Point Formats Extension, when used, MUST contain the value 0 (uncompressed) as one of the items in the list of point formats.

Actions of the receiver:

A client that receives a ServerHello message containing a Supported Point Formats Extension MUST respect the server's choice of point formats during the handshake (cf. Section 5.6 and Section 5.7). If no Supported Point Formats Extension is received with the ServerHello, this is equivalent to an extension allowing only the uncompressed point format.

5.3. Server Certificate

When this message is sent:

This message is sent in all non-anonymous ECC-based key exchange algorithms.

Meaning of this message:

This message is used to authentically convey the server's static public key to the client. The following table shows the server certificate type appropriate for each key exchange algorithm. ECC public keys MUST be encoded in certificates as described in Section 5.9.

NOTE: The server's Certificate message is capable of carrying a chain of certificates. The restrictions mentioned in Table 3 apply only to the server's certificate (first in the chain).

Algorithm	Server Certificate Type
ECDHE_ECDSA	Certificate MUST contain an ECDSA- or EdDSA-capable public key.
ECDHE_RSA	Certificate MUST contain an RSA public key.

Table 2: Server Certificate Types

Structure of this message:

Identical to the TLS Certificate format.

Actions of the sender:

The server constructs an appropriate certificate chain and conveys it to the client in the Certificate message. If the client has used a Supported Elliptic Curves Extension, the public key in the server's certificate MUST respect the client's choice of elliptic curves. A server that cannot satisfy this requirement MUST NOT choose an ECC cipher suite in its ServerHello message.)

Actions of the receiver:

The client validates the certificate chain, extracts the server's public key, and checks that the key type is appropriate for the negotiated key exchange algorithm. (A possible reason for a fatal handshake failure is that the client's capabilities for handling elliptic curves and point formats are exceeded; cf. Section 5.1.)

5.4. Server Key Exchange

When this message is sent:

This message is sent when using the ECDHE_ECDSA, ECDHE_RSA, and ECDH_anon key exchange algorithms.

Meaning of this message:

This message is used to convey the server's ephemeral ECDH public key (and the corresponding elliptic curve domain parameters) to the client.

The ECCCurveType enum used to have values for explicit prime and for explicit char2 curves. Those values are now deprecated, so only one value remains:

Structure of this message:

```
enum {
    deprecated (1..2),
    named_curve (3),
    reserved(248..255)
} ECCurveType;
```

The value named_curve indicates that a named curve is used. This option is now the only remaining format.

Values 248 through 255 are reserved for private use.

The ECCurveType name space is maintained by IANA. See Section 9 for information on how new value assignments are added.

RFC 4492 had a specification for an ECCurve structure and an ECBasisType structure. Both of these are omitted now because they were only used with the now deprecated explicit curves.

```
struct {
    opaque point <1..2^8-1>;
} ECPoint;
```

point: This is the byte string representation of an elliptic curve point following the conversion routine in Section 4.3.6 of [ANSI.X9-62.2005]. This byte string may represent an elliptic curve point in uncompressed, compressed, or hybrid format, but this specification deprecates all but the uncompressed format. For the NIST curves, the format is repeated in Section 5.4.1 for convenience. For the X25519 and X448 curves, the only valid representation is the one specified in [RFC7748] - a 32- or 56-octet representation of the u value of the point. This structure MUST NOT be used with Ed25519 and Ed448 public keys.

```
struct {
    ECCurveType    curve_type;
    select (curve_type) {
        case named_curve:
            NamedCurve namedcurve;
    };
} ECParameters;
```

curve_type: This identifies the type of the elliptic curve domain parameters.

namedCurve: Specifies a recommended set of elliptic curve domain parameters. All those values of NamedCurve are allowed that refer to a curve capable of Diffie-Hellman. With the deprecation of the explicit curves, this now includes all of the NamedCurve values.

```
        struct {
            ECParameters    curve_params;
            ECPoint         public;
        } ServerECDHParams;
```

curve_params: Specifies the elliptic curve domain parameters associated with the ECDH public key.

public: The ephemeral ECDH public key.

The ServerKeyExchange message is extended as follows.

```
        enum {
            ec_diffie_hellman
        } KeyExchangeAlgorithm;
```

- o ec_diffie_hellman: Indicates the ServerKeyExchange message contains an ECDH public key.

```
   select (KeyExchangeAlgorithm) {
       case ec_diffie_hellman:
           ServerECDHParams    params;
           Signature           signed_params;
   } ServerKeyExchange;
```

- o params: Specifies the ECDH public key and associated domain parameters.
- o signed_params: A hash of the params, with the signature appropriate to that hash applied. The private key corresponding to the certified public key in the server’s Certificate message is used for signing.

```
        enum {
            ecdsa(3),
            ed25519(TBD3)
            ed448(TBD4)
        } SignatureAlgorithm;
        select (SignatureAlgorithm) {
           case ecdsa:
                digitally-signed struct {
                    opaque sha_hash[sha_size];
                };
           case ed25519,ed448:
                digitally-signed struct {
                    opaque rawdata[rawdata_size];
                };
        } Signature;
      ServerKeyExchange.signed_params.sha_hash
          SHA(ClientHello.random + ServerHello.random +
                                 ServerKeyExchange.params);
      ServerKeyExchange.signed_params.rawdata
          ClientHello.random + ServerHello.random +
                                 ServerKeyExchange.params;
```

NOTE: SignatureAlgorithm is "rsa" for the ECDHE_RSA key exchange algorithm and "anonymous" for ECDH_anon. These cases are defined in TLS. SignatureAlgorithm is "ecdsa" or "eddsa" for ECDHE_ECDSA. ECDSA signatures are generated and verified as described in Section 5.10, and SHA in the above template for sha_hash accordingly may denote a hash algorithm other than SHA-1. As per ANSI X9.62, an ECDSA signature consists of a pair of integers, r and s. The digitally-signed element is encoded as an opaque vector <0..2^16-1>, the contents of which are the DER encoding corresponding to the following ASN.1 notation.

```
           Ecdsa-Sig-Value ::= SEQUENCE {
               r       INTEGER,
               s       INTEGER
           }
```

EdDSA signatures in both the protocol and in certificates that conform to [PKIX-EdDSA] are generated and verified according to [RFC8032]. The digitally-signed element is encoded as an opaque vector<0..2^16-1>, the contents of which is the octet string output of the EdDSA signing algorithm.

Actions of the sender:

The server selects elliptic curve domain parameters and an ephemeral ECDH public key corresponding to these parameters according to the

ECKAS-DH1 scheme from IEEE 1363 [IEEE.P1363.1998]. It conveys this information to the client in the ServerKeyExchange message using the format defined above.

Actions of the receiver:

The client verifies the signature (when present) and retrieves the server’s elliptic curve domain parameters and ephemeral ECDH public key from the ServerKeyExchange message. (A possible reason for a fatal handshake failure is that the client’s capabilities for handling elliptic curves and point formats are exceeded; cf. Section 5.1.)

5.4.1. Uncompressed Point Format for NIST curves

The following represents the wire format for representing ECPoint in ServerKeyExchange records. The first octet of the representation indicates the form, which may be compressed, uncompressed, or hybrid. This specification supports only the uncompressed format for these curves. This is followed by the binary representation of the X value in "big-endian" or "network" format, followed by the binary representation of the Y value in "big-endian" or "network" format. There are no internal length markers, so each number representation occupies as many octets as implied by the curve parameters. For P-256 this means that each of X and Y use 32 octets, padded on the left by zeros if necessary. For P-384 they take 48 octets each, and for P-521 they take 66 octets each.

Here’s a more formal representation:

```
enum {
    uncompressed(4),
    (255)
  } PointConversionForm;

struct {
    PointConversionForm  form;
    opaque               X[coordinate_length];
    opaque               Y[coordinate_length];
} UncompressedPointRepresentation;
```

5.5. Certificate Request

When this message is sent:

This message is sent when requesting client authentication.

Meaning of this message:

The server uses this message to suggest acceptable client authentication methods.

Structure of this message:

The TLS CertificateRequest message is extended as follows.

```
        enum {
            ecdsa_sign(64),
            deprecated1(65),  /* was rsa_fixed_ecdh */
            deprecated2(66),  /* was ecdsa_fixed_ecdh */
            (255)
        } ClientCertificateType;
```

- ecdsa_sign: Indicates that the server would like to use the corresponding client authentication method specified in Section 3.

Note that RFC 4492 also defined RSA and ECDSA certificates that included a fixed ECDH public key. These mechanisms saw very little implementation so this specification is deprecating them.

Actions of the sender:

The server decides which client authentication methods it would like to use, and conveys this information to the client using the format defined above.

Actions of the receiver:

The client determines whether it has a suitable certificate for use with any of the requested methods and whether to proceed with client authentication.

5.6. Client Certificate

When this message is sent:

This message is sent in response to a CertificateRequest when a client has a suitable certificate and has decided to proceed with client authentication. (Note that if the server has used a Supported Point Formats Extension, a certificate can only be considered suitable for use with the ECDSA_sign authentication method if the public key point specified in it is uncompressed, as that is the only point format still supported.

Meaning of this message:

This message is used to authentically convey the client’s static public key to the server. The following table summarizes what client certificate types are appropriate for the ECC-based client authentication mechanisms described in Section 3. ECC public keys must be encoded in certificates as described in Section 5.9.

NOTE: The client’s Certificate message is capable of carrying a chain of certificates. The restrictions mentioned in Table 4 apply only to the client’s certificate (first in the chain).

The certificate MUST contain an ECDSA- or EdDSA-capable public key.

Structure of this message:

Identical to the TLS client Certificate format.

Actions of the sender:

The client constructs an appropriate certificate chain, and conveys it to the server in the Certificate message.

Actions of the receiver:

The TLS server validates the certificate chain, extracts the client’s public key, and checks that the key type is appropriate for the client authentication method.

5.7. Client Key Exchange

When this message is sent:

This message is sent in all key exchange algorithms. It contains the client’s ephemeral ECDH public key.

Meaning of the message:

This message is used to convey ephemeral data relating to the key exchange belonging to the client (such as its ephemeral ECDH public key).

Structure of this message:

The TLS ClientKeyExchange message is extended as follows.

```
enum {
    implicit,
    explicit
} PublicValueEncoding;
```

- implicit, explicit: For ECC cipher suites, this indicates whether the client's ECDH public key is in the client's certificate ("implicit") or is provided, as an ephemeral ECDH public key, in the ClientKeyExchange message ("explicit"). The implicit encoding is deprecated and is retained here for backward compatibility only.

```
        struct {
            ECPoint ecdh_Yc;
        } ClientECDiffieHellmanPublic;
```

ecdh_Yc: Contains the client's ephemeral ECDH public key as a byte string ECPoint.point, which may represent an elliptic curve point in uncompressed format.

```
        struct {
            select (KeyExchangeAlgorithm) {
                case ec_diffie_hellman: ClientECDiffieHellmanPublic;
            } exchange_keys;
        } ClientKeyExchange;
```

Actions of the sender:

The client selects an ephemeral ECDH public key corresponding to the parameters it received from the server. The format is the same as in Section 5.4.

Actions of the receiver:

The server retrieves the client's ephemeral ECDH public key from the ClientKeyExchange message and checks that it is on the same elliptic curve as the server's ECDH key.

5.8. Certificate Verify

When this message is sent:

This message is sent when the client sends a client certificate containing a public key usable for digital signatures.

Meaning of the message:

This message contains a signature that proves possession of the private key corresponding to the public key in the client's Certificate message.

Structure of this message:

The TLS CertificateVerify message and the underlying Signature type are defined in the TLS base specifications, and the latter is extended here in Section 5.4. For the ecdsa and eddsa cases, the signature field in the CertificateVerify message contains an ECDSA or EdDSA (respectively) signature computed over handshake messages exchanged so far, exactly similar to CertificateVerify with other signing algorithms:

```
        CertificateVerify.signature.sha_hash
            SHA(handshake_messages);
        CertificateVerify.signature.rawdata
            handshake_messages;
```

ECDSA signatures are computed as described in Section 5.10, and SHA in the above template for sha_hash accordingly may denote a hash algorithm other than SHA-1. As per ANSI X9.62, an ECDSA signature consists of a pair of integers, r and s. The digitally-signed element is encoded as an opaque vector <0..2^16-1>, the contents of which are the DER encoding [CCITT.X690] corresponding to the following ASN.1 notation [CCITT.X680].

```
        Ecdsa-Sig-Value ::= SEQUENCE {
            r       INTEGER,
            s       INTEGER
        }
```

EdDSA signatures are generated and verified according to [RFC8032]. The digitally-signed element is encoded as an opaque vector<0..2^16-1>, the contents of which is the octet string output of the EdDSA signing algorithm.

Actions of the sender:

The client computes its signature over all handshake messages sent or received starting at client hello and up to but not including this message. It uses the private key corresponding to its certified public key to compute the signature, which is conveyed in the format defined above.

Actions of the receiver:

The server extracts the client's signature from the CertificateVerify message, and verifies the signature using the public key it received in the client's Certificate message.

5.9. Elliptic Curve Certificates

X.509 certificates containing ECC public keys or signed using ECDSA MUST comply with [RFC3279] or another RFC that replaces or extends it. X.509 certificates containing ECC public keys or signed using EdDSA MUST comply with [PKIX-EdDSA]. Clients SHOULD use the elliptic curve domain parameters recommended in ANSI X9.62, FIPS 186-4, and SEC 2 [SECG-SEC2] or in [RFC8032].

EdDSA keys using the Ed25519 algorithm MUST use the ed25519 signature algorithm, and Ed448 keys MUST use the ed448 signature algorithm. This document does not define use of Ed25519ph and Ed448ph keys with TLS. Ed25519, Ed25519ph, Ed448, and Ed448ph keys MUST NOT be used with ECDSA.

5.10. ECDH, ECDSA, and RSA Computations

All ECDH calculations for the NIST curves (including parameter and key generation as well as the shared secret calculation) are performed according to [IEEE.P1363.1998] using the ECKAS-DH1 scheme with the identity map as key derivation function (KDF), so that the premaster secret is the x-coordinate of the ECDH shared secret elliptic curve point represented as an octet string. Note that this octet string (Z in IEEE 1363 terminology) as output by FE2OSP, the Field Element to Octet String Conversion Primitive, has constant length for any given field; leading zeros found in this octet string MUST NOT be truncated.

(Note that this use of the identity KDF is a technicality. The complete picture is that ECDH is employed with a non-trivial KDF because TLS does not directly use the premaster secret for anything other than for computing the master secret. In TLS 1.0 and 1.1, this means that the MD5- and SHA-1-based TLS PRF serves as a KDF; in TLS 1.2 the KDF is determined by ciphersuite; it is conceivable that future TLS versions or new TLS extensions introduced in the future may vary this computation.)

An ECDHE key exchange using X25519 (curve x25519) goes as follows: Each party picks a secret key d uniformly at random and computes the corresponding public key x = X25519(d, G). Parties exchange their public keys, and compute a shared secret as x_S = X25519(d, x_peer). If either party obtains all-zeroes x_S, it MUST abort the handshake (as required by definition of X25519 and X448). ECDHE for X448 works similarily, replacing X25519 with X448, and x25519 with x448. The derived shared secret is used directly as the premaster secret, which is always exactly 32 bytes when ECDHE with X25519 is used and 56 bytes when ECDHE with X448 is used.

All ECDSA computations MUST be performed according to ANSI X9.62 or its successors. Data to be signed/verified is hashed, and the result run directly through the ECDSA algorithm with no additional hashing. A secure hash function such as SHA-256, SHA-384, or SHA-512 from [FIPS.180-4] MUST be used.

All EdDSA computations MUST be performed according to [RFC8032] or its succesors. Data to be signed/verified is run through the EdDSA algorithm wih no hashing (EdDSA will internally run the data through the PH function). The context parameter for Ed448 MUST be set to the empty string.

RFC 4492 anticipated the standardization of a mechanism for specifying the required hash function in the certificate, perhaps in the parameters field of the subjectPublicKeyInfo. Such standardization never took place, and as a result, SHA-1 is used in TLS 1.1 and earlier (except for EdDSA, which uses identity function). TLS 1.2 added a SignatureAndHashAlgorithm parameter to the DigitallySigned struct, thus allowing agility in choosing the signature hash. EdDSA signatures MUST have HashAlgorithm of TBD5 (Intrinsic).

All RSA signatures must be generated and verified according to [PKCS1] block type 1.

5.11. Public Key Validation

With the NIST curves, each party MUST validate the public key sent by its peer in the ClientKeyExchange and ServerKeyExchange messages. A receiving party MUST check that the x and y parameters from the peer's public value satisfy the curve equation, $y^2 = x^3 + ax + b \mod p$. See section 2.3 of [Menezes] for details. Failing to do so allows attackers to gain information about the private key, to the point that they may recover the entire private key in a few requests, if that key is not really ephemeral.

With X25519 and X448, a receiving party MUST check whether the computed premaster secret is the all-zero value and abort the handshake if so, as described in section 6 of [RFC7748].

Ed25519 and Ed448 internally do public key validation as part of signature verification.

6. Cipher Suites

The table below defines ECC cipher suites that use the key exchange algorithms specified in Section 2.

CipherSuite	Identifier
TLS_ECDHE_ECDSA_WITH_NULL_SHA	{ 0xC0, 0x06 }
TLS_ECDHE_ECDSA_WITH_3DES_EDE_CBC_SHA	{ 0xC0, 0x08 }
TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA	{ 0xC0, 0x09 }
TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA	{ 0xC0, 0x0A }
TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256	{ 0xC0, 0x2B }
TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384	{ 0xC0, 0x2C }
TLS_ECDHE_RSA_WITH_NULL_SHA	{ 0xC0, 0x10 }
TLS_ECDHE_RSA_WITH_3DES_EDE_CBC_SHA	{ 0xC0, 0x12 }
TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA	{ 0xC0, 0x13 }
TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA	{ 0xC0, 0x14 }
TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256	{ 0xC0, 0x2F }
TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384	{ 0xC0, 0x30 }
TLS_ECDH_anon_WITH_NULL_SHA	{ 0xC0, 0x15 }
TLS_ECDH_anon_WITH_3DES_EDE_CBC_SHA	{ 0xC0, 0x17 }
TLS_ECDH_anon_WITH_AES_128_CBC_SHA	{ 0xC0, 0x18 }
TLS_ECDH_anon_WITH_AES_256_CBC_SHA	{ 0xC0, 0x19 }

Table 3: TLS ECC cipher suites

The key exchange method, cipher, and hash algorithm for each of these cipher suites are easily determined by examining the name. Ciphers (other than AES ciphers) and hash algorithms are defined in [RFC2246] and [RFC4346]. AES ciphers are defined in [RFC5246], and AES-GCM ciphersuites are in [RFC5289].

Server implementations SHOULD support all of the following cipher suites, and client implementations SHOULD support at least one of them:

- o TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256
- o TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA
- o TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256
- o TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA256

7. Implementation Status

Both ECDHE and ECDSA with the NIST curves are widely implemented, supported in all major browsers and all widely used TLS libraries. ECDHE with Curve25519 is by now implemented in several browsers and several TLS libraries including OpenSSL. Curve448 and EdDSA have working, interoperable implementations, but are not yet as widely deployed.

8. Security Considerations

Security issues are discussed throughout this memo.

For TLS handshakes using ECC cipher suites, the security considerations in appendices D of all three TLS base documemts apply accordingly.

Security discussions specific to ECC can be found in [IEEE.P1363.1998] and [ANSI.X9-62.2005]. One important issue that implementers and users must consider is elliptic curve selection. Guidance on selecting an appropriate elliptic curve size is given in Table 1. Security considerations specific to X25519 and X448 are discussed in section 7 of [RFC7748].

Beyond elliptic curve size, the main issue is elliptic curve structure. As a general principle, it is more conservative to use elliptic curves with as little algebraic structure as possible. Thus, random curves are more conservative than special curves such as Koblitz curves, and curves over F_p with p random are more conservative than curves over F_p with p of a special form, and curves over F_p with p random are considered more conservative than curves over F_2^m as there is no choice between multiple fields of similar size for characteristic 2.

Another issue is the potential for catastrophic failures when a single elliptic curve is widely used. In this case, an attack on the elliptic curve might result in the compromise of a large number of keys. Again, this concern may need to be balanced against efficiency and interoperability improvements associated with widely-used curves. Substantial additional information on elliptic curve choice can be found in [IEEE.P1363.1998], [ANSI.X9-62.2005], and [FIPS.186-4].

The Introduction of [RFC8032] lists the security, performance, and operational advantages of EdDSA signatures over ECDSA signatures using the NIST curves.

All of the key exchange algorithms defined in this document provide forward secrecy. Some of the deprecated key exchange algorithms do not.

9. IANA Considerations

[RFC4492], the predecessor of this document has already defined the IANA registries for the following:

- o Supported Groups Section 5.1
- o ECPointFormat Section 5.1

- o ECCurveType Section 5.4

IANA is requested to prepend "TLS" to the names of the previous three registries.

For each name space, this document defines the initial value assignments and defines a range of 256 values (NamedCurve) or eight values (ECPointFormat and ECCurveType) reserved for Private Use. The policy for any additional assignments is "Specification Required". The previous version of this document required IETF review.

NOTE: IANA, please update the registries to reflect the new policy.

NOTE: RFC editor please delete these two notes prior to publication.

IANA, please update these two registries to refer to this document.

IANA is requested to assigned the value 29 to x25519, and the value 30 to x448 in the TLS Supported Groups Registry. This replaces the temporary registrations ecdh_x25519(29) and ecdh_x448(30).

IANA is requested to assign two values from the TLS SignatureAlgorithm Registry with names ed25519(TBD3) and ed448(TBD4) with this document as reference. To keep compatibility with TLS 1.3, TBD3 should be 7, and TBD4 should be 8.

IANA is requested to assign one value from the "TLS HashAlgorithm Registry" with name Intrinsic(TBD5) and this document as reference. To keep compatibility with TLS 1.3, TBD5 should be 8 and DTLS-OK should be set to true (Y).

10. Acknowledgements

Most of the text is this document is taken from [RFC4492], the predecessor of this document. The authors of that document were:

- o Simon Blake-Wilson
- o Nelson Bolyard
- o Vipul Gupta
- o Chris Hawk
- o Bodo Moeller

In the predecessor document, the authors acknowledged the contributions of Bill Anderson and Tim Dierks.

The author would like to thank Nikos Mavrogiannopoulos, Martin Thomson, and Tanja Lange for contributions to this document.

11. Version History for This Draft

NOTE TO RFC EDITOR: PLEASE REMOVE THIS SECTION

Changes from draft-ietf-tls-rfc4492bis-03 to draft-nir-tls-rfc4492bis-05:

- o Add support for CFRG curves and signatures work.

Changes from draft-ietf-tls-rfc4492bis-01 to draft-nir-tls-rfc4492bis-03:

- o Removed unused curves.
- o Removed unused point formats (all but uncompressed)

Changes from draft-nir-tls-rfc4492bis-00 and draft-ietf-tls-rfc4492bis-00 to draft-nir-tls-rfc4492bis-01:

- o Merged errata
- o Removed ECDH_RSA and ECDH_ECDSA

Changes from RFC 4492 to draft-nir-tls-rfc4492bis-00:

- o Added TLS 1.2 to references.
- o Moved RFC 4492 authors to acknowledgements.
- o Removed list of required reading for ECC.
- o Prepended "TLS" to the names of the three registries defined in the IANA Considerations section.

12. References

12.1. Normative References

[ANSI.X9-62.2005]
American National Standards Institute, "Public Key Cryptography for the Financial Services Industry, The Elliptic Curve Digital Signature Algorithm (ECDSA)", ANSI X9.62, 2005.

[CCITT.X680]
International Telephone and Telegraph Consultative Committee, "Abstract Syntax Notation One (ASN.1): Specification of basic notation", CCITT Recommendation X.680, July 2002.

[CCITT.X690]
International Telephone and Telegraph Consultative Committee, "ASN.1 encoding rules: Specification of basic encoding Rules (BER), Canonical encoding rules (CER) and Distinguished encoding rules (DER)", CCITT Recommendation X.690, July 2002.

[FIPS.186-4]
National Institute of Standards and Technology, "Digital Signature Standard", FIPS PUB 186-4, 2013, <http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf>.

[PKCS1] RSA Laboratories, "RSA Encryption Standard, Version 1.5", PKCS 1, November 1993.

[PKIX-EdDSA]
Josefsson, S. and J. Schaad, "Algorithm Identifiers for Ed25519, Ed25519ph, Ed448, Ed448ph, X25519 and X448 for use in the Internet X.509 Public Key Infrastructure", August 2016, <https://tools.ietf.org/html/draft-ietf-curdle-pkix-01>.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

[RFC2246] Dierks, T. and C. Allen, "The TLS Protocol Version 1.0", RFC 2246, January 1999.

[RFC3279] Bassham, L., Polk, W., and R. Housley, "Algorithms and Identifiers for the Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile", RFC 3279, April 2002.

[RFC4346] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.1", RFC 4346, April 2006.

[RFC4366] Blake-Wilson, S., Nystrom, M., Hopwood, D., Mikkelsen, J., and T. Wright, "Transport Layer Security (TLS) Extensions", RFC 4366, April 2006.

[RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, August 2008.

[RFC5289] Rescorla, E., "TLS Elliptic Curve Cipher Suites with SHA-256/384 and AES Galois Counter Mode (GCM)", RFC 5289, August 2008.

[RFC7748] Langley, A., Hamburg, M., and S. Turner, "Elliptic Curves for Security", RFC 7748, January 2016.

[RFC8032] Josefsson, S. and I. Liusvaara, "Edwards-Curve Digital Signature Algorithm (EdDSA)", RFC 8032, January 2017.

[SECG-SEC2] CECG, "Recommended Elliptic Curve Domain Parameters", SEC 2, 2000.

12.2. Informative References

[FIPS.180-4] National Institute of Standards and Technology, "Secure Hash Standard (SHS)", FIPS PUB 180-4, August 2015, <http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.180-4.pdf>.

[I-D.ietf-tls-tls13] Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", draft-ietf-tls-tls13-18 (work in progress), October 2016.

[IEEE.P1363.1998] Institute of Electrical and Electronics Engineers, "Standard Specifications for Public Key Cryptography", IEEE Draft P1363, 1998.

[Menezes] Menezes, A. and B. Ustaoglu, "On Reusing Ephemeral Keys In Diffie-Hellman Key Agreement Protocols", IACR Menezes2008, December 2008.

[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, May 2006.

[RFC7919] Gillmor, D., "Negotiated Finite Field Diffie-Hellman Ephemeral Parameters for Transport Layer Security (TLS)", RFC 7919, DOI 10.17487/RFC7919, August 2016, <http://www.rfc-editor.org/info/rfc7919>.

Appendix A. Equivalent Curves (Informative)

All of the NIST curves [FIPS.186-4] and several of the ANSI curves [ANSI.X9-62.2005] are equivalent to curves listed in Section 5.1.1. In the following table, multiple names in one row represent aliases for the same curve.

Curve names chosen by different standards organizations

SECG	ANSI X9.62	NIST
sect163k1		NIST K-163
sect163r1		
sect163r2		NIST B-163
sect193r1		
sect193r2		
sect233k1		NIST K-233
sect233r1		NIST B-233
sect239k1		
sect283k1		NIST K-283
sect283r1		NIST B-283
sect409k1		NIST K-409
sect409r1		NIST B-409
sect571k1		NIST K-571
sect571r1		NIST B-571
secp160k1		
secp160r1		
secp160r2		
secp192k1		
secp192r1	prime192v1	NIST P-192
secp224k1		
secp224r1		NIST P-224
secp256k1		
secp256r1	prime256v1	NIST P-256
secp384r1		NIST P-384
secp521r1		NIST P-521

Table 4: Equivalent curves defined by SECG, ANSI, and NIST

Appendix B. Differences from RFC 4492

- Added TLS 1.2
- Merged Errata
- Removed the ECDH key exchange algorithms: ECDH_RSA and ECDH_ECDSA
- Deprecated a bunch of ciphersuites:

    TLS_ECDH_ECDSA_WITH_NULL_SHA
    TLS_ECDH_ECDSA_WITH_RC4_128_SHA
    TLS_ECDH_ECDSA_WITH_3DES_EDE_CBC_SHA
    TLS_ECDH_ECDSA_WITH_AES_128_CBC_SHA
    TLS_ECDH_ECDSA_WITH_AES_256_CBC_SHA
    TLS_ECDH_RSA_WITH_NULL_SHA
    TLS_ECDH_RSA_WITH_RC4_128_SHA

TLS_ECDH_RSA_WITH_3DES_EDE_CBC_SHA
TLS_ECDH_RSA_WITH_AES_128_CBC_SHA
TLS_ECDH_RSA_WITH_AES_256_CBC_SHA
All the other RC4 ciphersuites

Removed unused curves and all but the uncompressed point format.

Added X25519 and X448.

Deprecated explicit curves.

Removed restriction on signature algorithm in certificate.

Authors' Addresses

Yoav Nir
Check Point Software Technologies Ltd.
5 Hasolelim st.
Tel Aviv 6789735
Israel

Email: ynir.ietf@gmail.com

Simon Josefsson
SJD AB

Email: simon@josefsson.org

Manuel Pegourie-Gonnard
Independent / PolarSSL

Email: mpg@elzevir.fr

Network Working Group   E. Rescorla
Internet-Draft   RTFM, Inc.
Obsoletes: 5077, 5246, 6961 (if   March 20, 2018
approved)
Updates: 4492, 5705, 6066 (if approved)
Intended status: Standards Track
Expires: September 21, 2018

The Transport Layer Security (TLS) Protocol Version 1.3
draft-ietf-tls-tls13-28

Abstract

This document specifies version 1.3 of the Transport Layer Security (TLS) protocol. TLS allows client/server applications to communicate over the Internet in a way that is designed to prevent eavesdropping, tampering, and message forgery.

This document updates RFCs 4492, 5705, and 6066 and it obsoletes RFCs 5077, 5246, and 6961. This document also specifies new requirements for TLS 1.2 implementations.

Table of Contents

1. Introduction

RFC EDITOR: PLEASE REMOVE THE FOLLOWING PARAGRAPH The source for this draft is maintained in GitHub. Suggested changes should be submitted as pull requests at https://github.com/tlswg/tls13-spec. Instructions are on that page as well. Editorial changes can be managed in GitHub, but any substantive change should be discussed on the TLS mailing list.

The primary goal of TLS is to provide a secure channel between two communicating peers; the only requirement from the underlying transport is a reliable, in-order, data stream. Specifically, the secure channel should provide the following properties:

- Authentication: The server side of the channel is always authenticated; the client side is optionally authenticated. Authentication can happen via asymmetric cryptography (e.g., RSA [RSA], ECDSA [ECDSA], EdDSA [RFC8032]) or a pre-shared key (PSK).

- Confidentiality: Data sent over the channel after establishment is only visible to the endpoints. TLS does not hide the length of the data it transmits, though endpoints are able to pad TLS records in order to obscure lengths and improve protection against traffic analysis techniques.

- Integrity: Data sent over the channel after establishment cannot be modified by attackers.

These properties should be true even in the face of an attacker who has complete control of the network, as described in [RFC3552]. See Appendix E for a more complete statement of the relevant security properties.

TLS consists of two primary components:

- A handshake protocol (Section 4) that authenticates the communicating parties, negotiates cryptographic modes and parameters, and establishes shared keying material. The handshake protocol is designed to resist tampering; an active attacker should not be able to force the peers to negotiate different parameters than they would if the connection were not under attack.

- A record protocol (Section 5) that uses the parameters established by the handshake protocol to protect traffic between the communicating peers. The record protocol divides traffic up into a series of records, each of which is independently protected using the traffic keys.

TLS is application protocol independent; higher-level protocols can layer on top of TLS transparently. The TLS standard, however, does not specify how protocols add security with TLS; how to initiate TLS handshaking and how to interpret the authentication certificates exchanged are left to the judgment of the designers and implementors of protocols that run on top of TLS.

This document defines TLS version 1.3. While TLS 1.3 is not directly compatible with previous versions, all versions of TLS incorporate a versioning mechanism which allows clients and servers to interoperably negotiate a common version if one is supported by both peers.

This document supersedes and obsoletes previous versions of TLS including version 1.2 [RFC5246]. It also obsoletes the TLS ticket mechanism defined in [RFC5077] and replaces it with the mechanism defined in Section 2.2. Section 4.2.7 updates [RFC4492] by modifying the protocol attributes used to negotiate Elliptic Curves. Because TLS 1.3 changes the way keys are derived, it updates [RFC5705] as described in Section 7.5. It also changes how OCSP messages are carried and therefore updates [RFC6066] and obsoletes [RFC6961] as described in section Section 4.4.2.1.

1.1. Conventions and Terminology

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.

The following terms are used:

client: The endpoint initiating the TLS connection.

connection: A transport-layer connection between two endpoints.

endpoint: Either the client or server of the connection.

handshake: An initial negotiation between client and server that establishes the parameters of their subsequent interactions within TLS.

peer: An endpoint. When discussing a particular endpoint, "peer" refers to the endpoint that is not the primary subject of discussion.

receiver: An endpoint that is receiving records.

sender: An endpoint that is transmitting records.

server: The endpoint which did not initiate the TLS connection.

1.2. Change Log

RFC EDITOR PLEASE DELETE THIS SECTION.

(*) indicates changes to the wire protocol which may require implementations to update.

draft-28

Add a section on exposure of PSK identities.

draft-27

- SHOULD->MUST for being able to process "supported_versions" without 0x0304.

- Much editorial cleanup.

draft-26

- Clarify that you can't negotiate pre-TLS 1.3 with supported_versions.

draft-25

- Add the header to additional data (*)

- Minor clarifications.

- IANA cleanup.

draft-24

- Require that CH2 have version 0303 (*)

- Some clarifications

draft-23

- Renumber key_share (*)

- Add a new extension and new code points to allow negotiating PSS separately for certificates and CertificateVerify (*)

- Slightly restrict when CCS must be accepted to make implementation easier.

- Document protocol invariants

- Add some text on the security of static RSA.

draft-22

- Implement changes for improved middlebox penetration (*)

- Move server_certificate_type to encrypted extensions (*)

- Allow resumption with a different SNI (*)

- Padding extension can change on HRR (*)

- Allow an empty ticket_nonce (*)

- Remove requirement to immediately respond to close_notify with close_notify (allowing half-close)

draft-21

- Add a per-ticket nonce so that each ticket is associated with a different PSK (*).

- Clarify that clients should send alerts with the handshake key if possible.

- Update state machine to show rekeying events

- Add discussion of 0-RTT and replay. Recommend that implementations implement some anti-replay mechanism.

draft-20

- Add "post_handshake_auth" extension to negotiate post-handshake authentication (*).

- Shorten labels for HKDF-Expand-Label so that we can fit within one compression block (*).

- Define how RFC 7250 works (*).

- Re-enable post-handshake client authentication even when you do PSK. The previous prohibition was editorial error.

- Remove cert_type and user_mapping, which don’t work on TLS 1.3 anyway.

- Added the no_application_protocol alert from [RFC7301] to the list of extensions.

- Added discussion of traffic analysis and side channel attacks.

draft-19

- Hash context_value input to Exporters (*)

- Add an additional Derive-Secret stage to Exporters (*).

- Hash ClientHello1 in the transcript when HRR is used. This reduces the state that needs to be carried in cookies. (*)

- Restructure CertificateRequest to have the selectors in extensions. This also allowed defining a "certificate_authorities" extension which can be used by the client instead of trusted_ca_keys (*).

- Tighten record framing requirements and require checking of them (*).

- Consolidate "ticket_early_data_info" and "early_data" into a single extension (*).

- Change end_of_early_data to be a handshake message (*).

- Add pre-extract Derive-Secret stages to key schedule (*).

- Remove spurious requirement to implement "pre_shared_key".

- Clarify location of "early_data" from server (it goes in EE, as indicated by the table in S 10).

- Require peer public key validation

- Add state machine diagram.

draft-18

- Remove unnecessary resumption_psk which is the only thing expanded from the resumption master secret. (*).

- Fix signature_algorithms entry in extensions table.

- Restate rule from RFC 6066 that you can’t resume unless SNI is the same.

draft-17

- Remove 0-RTT Finished and resumption_context, and replace with a psk_binder field in the PSK itself (*)

- Restructure PSK key exchange negotiation modes (*)

- Add max_early_data_size field to TicketEarlyDataInfo (*)

- Add a 0-RTT exporter and change the transcript for the regular exporter (*)

- Merge TicketExtensions and Extensions registry. Changes ticket_early_data_info code point (*)

- Replace Client.key_shares in response to HRR (*)

- Remove redundant labels for traffic key derivation (*)

- Harmonize requirements about cipher suite matching: for resumption you need to match KDF but for 0-RTT you need whole cipher suite. This allows PSKs to actually negotiate cipher suites. (*)

- Move SCT and OCSP into Certificate.extensions (*)

- Explicitly allow non-offered extensions in NewSessionTicket

- Explicitly allow predicting client Finished for NST

- Clarify conditions for allowing 0-RTT with PSK

draft-16

- Revise version negotiation (*)

- Change RSASSA-PSS and EdDSA SignatureScheme codepoints for better backwards compatibility (*)

- Move HelloRetryRequest.selected_group to an extension (*)

- Clarify the behavior of no exporter context and make it the same as an empty context.(*)

- New KeyUpdate format that allows for requesting/not-requesting an answer. This also means changes to the key schedule to support independent updates (*)

- New certificate_required alert (*)

- Forbid CertificateRequest with 0-RTT and PSK.

- Relax requirement to check SNI for 0-RTT.

draft-15

- New negotiation syntax as discussed in Berlin (*)

- Require CertificateRequest.context to be empty during handshake (*)

- Forbid empty tickets (*)

- Forbid application data messages in between post-handshake messages from the same flight (*)

- Clean up alert guidance (*)

- Clearer guidance on what is needed for TLS 1.2.

- Guidance on 0-RTT time windows.

- Rename a bunch of fields.

- Remove old PRNG text.

- Explicitly require checking that handshake records not span key changes.

draft-14

- Allow cookies to be longer (*)

- Remove the "context" from EarlyDataIndication as it was undefined and nobody used it (*)

- Remove 0-RTT EncryptedExtensions and replace the ticket_age extension with an obfuscated version. Also necessitates a change to NewSessionTicket (*).

- Move the downgrade sentinel to the end of ServerHello.Random to accommodate tlsdate (*).

- Define ecdsa_sha1 (*).
- Allow resumption even after fatal alerts. This matches current practice.
- Remove non-closure warning alerts. Require treating unknown alerts as fatal.
- Make the rules for accepting 0-RTT less restrictive.
- Clarify 0-RTT backward-compatibility rules.
- Clarify how 0-RTT and PSK identities interact.
- Add a section describing the data limits for each cipher.
- Major editorial restructuring.
- Replace the Security Analysis section with a WIP draft.

draft-13

- Allow server to send SupportedGroups.
- Remove 0-RTT client authentication
- Remove (EC)DHE 0-RTT.
- Flesh out 0-RTT PSK mode and shrink EarlyDataIndication
- Turn PSK-resumption response into an index to save room
- Move CertificateStatus to an extension
- Extra fields in NewSessionTicket.
- Restructure key schedule and add a resumption_context value.
- Require DH public keys and secrets to be zero-padded to the size of the group.
- Remove the redundant length fields in KeyShareEntry.
- Define a cookie field for HRR.

draft-12

- Provide a list of the PSK cipher suites.

- Remove the ability for the ServerHello to have no extensions (this aligns the syntax with the text).

- Clarify that the server can send application data after its first flight (0.5 RTT data)

- Revise signature algorithm negotiation to group hash, signature algorithm, and curve together. This is backwards compatible.

- Make ticket lifetime mandatory and limit it to a week.

- Make the purpose strings lower-case. This matches how people are implementing for interop.

- Define exporters.

- Editorial cleanup

draft-11

- Port the CFRG curves & signatures work from RFC4492bis.

- Remove sequence number and version from additional_data, which is now empty.

- Reorder values in HkdfLabel.

- Add support for version anti-downgrade mechanism.

- Update IANA considerations section and relax some of the policies.

- Unify authentication modes. Add post-handshake client authentication.

- Remove early_handshake content type. Terminate 0-RTT data with an alert.

- Reset sequence number upon key change (as proposed by Fournet et al.)

draft-10

- Remove ClientCertificateTypes field from CertificateRequest and add extensions.

- Merge client and server key shares into a single extension.

draft-09

- Change to RSA-PSS signatures for handshake messages.

- Remove support for DSA.

- Update key schedule per suggestions by Hugo, Hoeteck, and Bjoern Tackmann.

- Add support for per-record padding.

- Switch to encrypted record ContentType.

- Change HKDF labeling to include protocol version and value lengths.

- Shift the final decision to abort a handshake due to incompatible certificates to the client rather than having servers abort early.

- Deprecate SHA-1 with signatures.

- Add MTI algorithms.

draft-08

- Remove support for weak and lesser used named curves.

- Remove support for MD5 and SHA-224 hashes with signatures.

- Update lists of available AEAD cipher suites and error alerts.

- Reduce maximum permitted record expansion for AEAD from 2048 to 256 octets.

- Require digital signatures even when a previous configuration is used.

- Merge EarlyDataIndication and KnownConfiguration.

- Change code point for server_configuration to avoid collision with server_hello_done.

- Relax certificate_list ordering requirement to match current practice.

draft-07

- Integration of semi-ephemeral DH proposal.

- Add initial 0-RTT support.

- Remove resumption and replace with PSK + tickets.

- Move ClientKeyShare into an extension.

- Move to HKDF.

draft-06

- Prohibit RC4 negotiation for backwards compatibility.

- Freeze & deprecate record layer version field.

- Update format of signatures with context.

- Remove explicit IV.

draft-05

- Prohibit SSL negotiation for backwards compatibility.

- Fix which MS is used for exporters.

draft-04

- Modify key computations to include session hash.

- Remove ChangeCipherSpec.

- Renumber the new handshake messages to be somewhat more consistent with existing convention and to remove a duplicate registration.

- Remove renegotiation.

- Remove point format negotiation.

draft-03

- Remove GMT time.

- Merge in support for ECC from RFC 4492 but without explicit curves.

- Remove the unnecessary length field from the AD input to AEAD ciphers.

- Rename {Client,Server}KeyExchange to {Client,Server}KeyShare.

- Add an explicit HelloRetryRequest to reject the client's.

draft-02

- Increment version number.

- Rework handshake to provide 1-RTT mode.

- Remove custom DHE groups.

- Remove support for compression.

- Remove support for static RSA and DH key exchange.

- Remove support for non-AEAD ciphers.

1.3. Major Differences from TLS 1.2

The following is a list of the major functional differences between TLS 1.2 and TLS 1.3. It is not intended to be exhaustive and there are many minor differences.

- The list of supported symmetric algorithms has been pruned of all algorithms that are considered legacy. Those that remain all use Authenticated Encryption with Associated Data (AEAD) algorithms. The ciphersuite concept has been changed to separate the authentication and key exchange mechanisms from the record protection algorithm (including secret key length) and a hash to be used with the key derivation function and HMAC.

- A 0-RTT mode was added, saving a round-trip at connection setup for some application data, at the cost of certain security properties.

- Static RSA and Diffie-Hellman cipher suites have been removed; all public-key based key exchange mechanisms now provide forward secrecy.

- All handshake messages after the ServerHello are now encrypted. The newly introduced EncryptedExtension message allows various extensions previously sent in clear in the ServerHello to also enjoy confidentiality protection from active attackers.

- The key derivation functions have been re-designed. The new design allows easier analysis by cryptographers due to their improved key separation properties. The HMAC-based Extract-and-Expand Key Derivation Function (HKDF) is used as an underlying primitive.

- The handshake state machine has been significantly restructured to be more consistent and to remove superfluous messages such as ChangeCipherSpec (except when needed for middlebox compatibility).

- Elliptic curve algorithms are now in the base spec and new signature algorithms, such as ed25519 and ed448, are included. TLS 1.3 removed point format negotiation in favor of a single point format for each curve.

- Other cryptographic improvements including the removal of compression and custom DHE groups, changing the RSA padding to use RSASSA-PSS, and the removal of DSA.

- The TLS 1.2 version negotiation mechanism has been deprecated in favor of a version list in an extension. This increases compatibility with existing servers that incorrectly implemented version negotiation.

- Session resumption with and without server-side state as well as the PSK-based ciphersuites of earlier TLS versions have been replaced by a single new PSK exchange.

- Updated references to point to the updated versions of RFCs, as appropriate (e.g., RFC 5280 rather than RFC 3280).

1.4. Updates Affecting TLS 1.2

This document defines several changes that optionally affect implementations of TLS 1.2, including those which do not also support TLS 1.3:

- A version downgrade protection mechanism is described in Section 4.1.3.

- RSASSA-PSS signature schemes are defined in Section 4.2.3.

- The "supported_versions" ClientHello extension can be used to negotiate the version of TLS to use, in preference to the legacy_version field of the ClientHello.

- The "signature_algorithms_cert" extension allows a client to indicate which signature algorithms it can validate in X.509 certificates

Additionally, this document clarifies some compliance requirements for earlier versions of TLS; see Section 9.3.

2. Protocol Overview

The cryptographic parameters used by the secure channel are produced by the TLS handshake protocol. This sub-protocol of TLS is used by the client and server when first communicating with each other. The handshake protocol allows peers to negotiate a protocol version, select cryptographic algorithms, optionally authenticate each other, and establish shared secret keying material. Once the handshake is complete, the peers use the established keys to protect the application layer traffic.

A failure of the handshake or other protocol error triggers the termination of the connection, optionally preceded by an alert message (Section 6).

TLS supports three basic key exchange modes:

- (EC)DHE (Diffie-Hellman over either finite fields or elliptic curves)

- PSK-only

- PSK with (EC)DHE

Figure 1 below shows the basic full TLS handshake:

```
       Client                                           Server

Key  ^ ClientHello
Exch | + key_share*
     | + signature_algorithms*
     | + psk_key_exchange_modes*
     v + pre_shared_key*       -------->
                                                  ServerHello  ^ Key
                                                 + key_share*  | Exch
                                            + pre_shared_key*  v
                                        {EncryptedExtensions}  ^  Server
                                        {CertificateRequest*}  v  Params
                                               {Certificate*}  ^
                                         {CertificateVerify*}  | Auth
                                                   {Finished}  v
                               <--------  [Application Data*]
     ^ {Certificate*}
Auth | {CertificateVerify*}
     v {Finished}              -------->
       [Application Data]      <------->  [Application Data]

              +  Indicates noteworthy extensions sent in the
                 previously noted message.

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

              {} Indicates messages protected using keys
                 derived from a [sender]_handshake_traffic_secret.

              [] Indicates messages protected using keys
                 derived from [sender]_application_traffic_secret_N
```

Figure 1: Message flow for full TLS Handshake

The handshake can be thought of as having three phases (indicated in the diagram above):

- Key Exchange: Establish shared keying material and select the cryptographic parameters. Everything after this phase is encrypted.

- Server Parameters: Establish other handshake parameters (whether the client is authenticated, application layer protocol support, etc.).

- Authentication: Authenticate the server (and optionally the client) and provide key confirmation and handshake integrity.

In the Key Exchange phase, the client sends the ClientHello (Section 4.1.2) message, which contains a random nonce (ClientHello.random); its offered protocol versions; a list of symmetric cipher/HKDF hash pairs; either a set of Diffie-Hellman key shares (in the "key_share" extension Section 4.2.8), a set of pre-shared key labels (in the "pre_shared_key" extension Section 4.2.11) or both; and potentially additional extensions. Additional fields and/or messages may also be present for middlebox compatibility.

The server processes the ClientHello and determines the appropriate cryptographic parameters for the connection. It then responds with its own ServerHello (Section 4.1.3), which indicates the negotiated connection parameters. The combination of the ClientHello and the ServerHello determines the shared keys. If (EC)DHE key establishment is in use, then the ServerHello contains a "key_share" extension with the server's ephemeral Diffie-Hellman share; the server's share MUST be in the same group as one of the client's shares. If PSK key establishment is in use, then the ServerHello contains a "pre_shared_key" extension indicating which of the client's offered PSKs was selected. Note that implementations can use (EC)DHE and PSK together, in which case both extensions will be supplied.

The server then sends two messages to establish the Server Parameters:

EncryptedExtensions: responses to ClientHello extensions that are not required to determine the cryptographic parameters, other than those that are specific to individual certificates. [Section 4.3.1]

CertificateRequest: if certificate-based client authentication is desired, the desired parameters for that certificate. This message is omitted if client authentication is not desired. [Section 4.3.2]

Finally, the client and server exchange Authentication messages. TLS uses the same set of messages every time that certificate-based authentication is needed. (PSK-based authentication happens as a side effect of key exchange.) Specifically:

Certificate: the certificate of the endpoint and any per-certificate extensions. This message is omitted by the server if not authenticating with a certificate and by the client if the server did not send CertificateRequest (thus indicating that the client should not authenticate with a certificate). Note that if raw public keys [RFC7250] or the cached information extension [RFC7924] are in use, then this message will not contain a certificate but rather some other value corresponding to the server’s long-term key. [Section 4.4.2]

CertificateVerify: a signature over the entire handshake using the private key corresponding to the public key in the Certificate message. This message is omitted if the endpoint is not authenticating via a certificate. [Section 4.4.3]

Finished: a MAC (Message Authentication Code) over the entire handshake. This message provides key confirmation, binds the endpoint’s identity to the exchanged keys, and in PSK mode also authenticates the handshake. [Section 4.4.4]

Upon receiving the server’s messages, the client responds with its Authentication messages, namely Certificate and CertificateVerify (if requested), and Finished.

At this point, the handshake is complete, and the client and server derive the keying material required by the record layer to exchange application-layer data protected through authenticated encryption. Application data MUST NOT be sent prior to sending the Finished message, except as specified in [Section 2.3]. Note that while the server may send application data prior to receiving the client’s Authentication messages, any data sent at that point is, of course, being sent to an unauthenticated peer.

2.1. Incorrect DHE Share

If the client has not provided a sufficient "key_share" extension (e.g., it includes only DHE or ECDHE groups unacceptable to or unsupported by the server), the server corrects the mismatch with a HelloRetryRequest and the client needs to restart the handshake with an appropriate "key_share" extension, as shown in Figure 2. If no common cryptographic parameters can be negotiated, the server MUST abort the handshake with an appropriate alert.

```
         Client                                               Server

         ClientHello
         + key_share             -------->
                                                   HelloRetryRequest
                                 <--------               + key_share
         ClientHello
         + key_share             -------->
                                                         ServerHello
                                                         + key_share
                                               {EncryptedExtensions}
                                               {CertificateRequest*}
                                                      {Certificate*}
                                                {CertificateVerify*}
                                                          {Finished}
                                 <--------       [Application Data*]
         {Certificate*}
         {CertificateVerify*}
         {Finished}              -------->
         [Application Data]      <------->        [Application Data]
```

Figure 2: Message flow for a full handshake with mismatched parameters

Note: The handshake transcript incorporates the initial ClientHello/ HelloRetryRequest exchange; it is not reset with the new ClientHello.

TLS also allows several optimized variants of the basic handshake, as described in the following sections.

2.2. Resumption and Pre-Shared Key (PSK)

Although TLS PSKs can be established out of band, PSKs can also be established in a previous connection and then used to establish a new connection ("session resumption" or "resuming" with a PSK). Once a handshake has completed, the server can send to the client a PSK identity that corresponds to a unique key derived from the initial handshake (see Section 4.6.1). The client can then use that PSK identity in future handshakes to negotiate the use of the associated PSK. If the server accepts the PSK, then the security context of the new connection is cryptographically tied to the original connection and the key derived from the initial handshake is used to bootstrap the cryptographic state instead of a full handshake. In TLS 1.2 and below, this functionality was provided by "session IDs" and "session tickets" [RFC5077]. Both mechanisms are obsoleted in TLS 1.3.

PSKs can be used with (EC)DHE key exchange in order to provide forward secrecy in combination with shared keys, or can be used alone, at the cost of losing forward secrecy for the application data.

Figure 3 shows a pair of handshakes in which the first establishes a PSK and the second uses it:

```
       Client                                               Server

Initial Handshake:
       ClientHello
       + key_share               -------->
                                                       ServerHello
                                                       + key_share
                                             {EncryptedExtensions}
                                             {CertificateRequest*}
                                                    {Certificate*}
                                              {CertificateVerify*}
                                                        {Finished}
                                 <--------     [Application Data*]
       {Certificate*}
       {CertificateVerify*}
       {Finished}                -------->
                                 <--------      [NewSessionTicket]
       [Application Data]        <------->      [Application Data]


Subsequent Handshake:
       ClientHello
       + key_share*
       + pre_shared_key          -------->
                                                       ServerHello
                                                  + pre_shared_key
                                                      + key_share*
                                             {EncryptedExtensions}
                                                        {Finished}
                                 <--------     [Application Data*]
       {Finished}                -------->
       [Application Data]        <------->      [Application Data]
```

Figure 3: Message flow for resumption and PSK

As the server is authenticating via a PSK, it does not send a Certificate or a CertificateVerify message. When a client offers resumption via PSK, it SHOULD also supply a "key_share" extension to the server to allow the server to decline resumption and fall back to a full handshake, if needed. The server responds with a "pre_shared_key" extension to negotiate use of PSK key establishment and can (as shown here) respond with a "key_share" extension to do (EC)DHE key establishment, thus providing forward secrecy.

When PSKs are provisioned out of band, the PSK identity and the KDF hash algorithm to be used with the PSK MUST also be provisioned.

Note: When using an out-of-band provisioned pre-shared secret, a critical consideration is using sufficient entropy during the key generation, as discussed in [RFC4086]. Deriving a shared secret from a password or other low-entropy sources is not secure. A low-entropy secret, or password, is subject to dictionary attacks based on the PSK binder. The specified PSK authentication is not a strong password-based authenticated key exchange even when used with Diffie-Hellman key establishment. Specifically, it does not prevent an attacker that can observe the handshake from performing a brute-force attack on the password/pre-shared key.

2.3. 0-RTT Data

When clients and servers share a PSK (either obtained externally or via a previous handshake), TLS 1.3 allows clients to send data on the first flight ("early data"). The client uses the PSK to authenticate the server and to encrypt the early data.

As shown in Figure 4, the 0-RTT data is just added to the 1-RTT handshake in the first flight. The rest of the handshake uses the same messages as for a 1-RTT handshake with PSK resumption.

```
         Client                                               Server

         ClientHello
         + early_data
         + key_share*
         + psk_key_exchange_modes
         + pre_shared_key
         (Application Data*)     -------->
                                                         ServerHello
                                                    + pre_shared_key
                                                        + key_share*
                                               {EncryptedExtensions}
                                                       + early_data*
                                                          {Finished}
                                 <--------       [Application Data*]
         (EndOfEarlyData)
         {Finished}              -------->
         [Application Data]      <------->        [Application Data]

               +  Indicates noteworthy extensions sent in the
                  previously noted message.

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

               () Indicates messages protected using keys
                  derived from client_early_traffic_secret.

               {} Indicates messages protected using keys
                  derived from a [sender]_handshake_traffic_secret.

               [] Indicates messages protected using keys
                  derived from [sender]_application_traffic_secret_N
```

Figure 4: Message flow for a zero round trip handshake

IMPORTANT NOTE: The security properties for 0-RTT data are weaker than those for other kinds of TLS data. Specifically:

1. This data is not forward secret, as it is encrypted solely under keys derived using the offered PSK.

2. There are no guarantees of non-replay between connections. Protection against replay for ordinary TLS 1.3 1-RTT data is provided via the server’s Random value, but 0-RTT data does not depend on the ServerHello and therefore has weaker guarantees. This is especially relevant if the data is authenticated either with TLS client authentication or inside the application

protocol. The same warnings apply to any use of the early_exporter_master_secret.

0-RTT data cannot be duplicated within a connection (i.e., the server will not process the same data twice for the same connection) and an attacker will not be able to make 0-RTT data appear to be 1-RTT data (because it is protected with different keys.) Appendix E.5 contains a description of potential attacks and Section 8 describes mechanisms which the server can use to limit the impact of replay.

3. Presentation Language

This document deals with the formatting of data in an external representation. The following very basic and somewhat casually defined presentation syntax will be used.

3.1. Basic Block Size

The representation of all data items is explicitly specified. The basic data block size is one byte (i.e., 8 bits). Multiple byte data items are concatenations of bytes, from left to right, from top to bottom. From the byte stream, a multi-byte item (a numeric in the example) is formed (using C notation) by:

```
value = (byte[0] << 8*(n-1)) | (byte[1] << 8*(n-2)) |
        ... | byte[n-1];
```

This byte ordering for multi-byte values is the commonplace network byte order or big-endian format.

3.2. Miscellaneous

Comments begin with "/*" and end with "*/".

Optional components are denoted by enclosing them in "[[ ]]" double brackets.

Single-byte entities containing uninterpreted data are of type opaque.

A type alias T’ for an existing type T is defined by:

```
T T’;
```

3.3. Numbers

The basic numeric data type is an unsigned byte (uint8). All larger numeric data types are formed from fixed-length series of bytes concatenated as described in Section 3.1 and are also unsigned. The following numeric types are predefined.

```
uint8 uint16[2];
uint8 uint24[3];
uint8 uint32[4];
uint8 uint64[8];
```

All values, here and elsewhere in the specification, are transmitted in network byte (big-endian) order; the uint32 represented by the hex bytes 01 02 03 04 is equivalent to the decimal value 16909060.

3.4. Vectors

A vector (single-dimensioned array) is a stream of homogeneous data elements. The size of the vector may be specified at documentation time or left unspecified until runtime. In either case, the length declares the number of bytes, not the number of elements, in the vector. The syntax for specifying a new type, T', that is a fixed-length vector of type T is

```
T T'[n];
```

Here, T' occupies n bytes in the data stream, where n is a multiple of the size of T. The length of the vector is not included in the encoded stream.

In the following example, Datum is defined to be three consecutive bytes that the protocol does not interpret, while Data is three consecutive Datum, consuming a total of nine bytes.

```
opaque Datum[3];      /* three uninterpreted bytes */
Datum Data[9];        /* 3 consecutive 3-byte vectors */
```

Variable-length vectors are defined by specifying a subrange of legal lengths, inclusively, using the notation <floor..ceiling>. When these are encoded, the actual length precedes the vector's contents in the byte stream. The length will be in the form of a number consuming as many bytes as required to hold the vector's specified maximum (ceiling) length. A variable-length vector with an actual length field of zero is referred to as an empty vector.

```
T T'<floor..ceiling>;
```

In the following example, mandatory is a vector that must contain between 300 and 400 bytes of type opaque. It can never be empty. The actual length field consumes two bytes, a uint16, which is sufficient to represent the value 400 (see Section 3.3). Similarly, longer can represent up to 800 bytes of data, or 400 uint16 elements, and it may be empty. Its encoding will include a two-byte actual length field prepended to the vector. The length of an encoded vector must be an exact multiple of the length of a single element (e.g., a 17-byte vector of uint16 would be illegal).

```
opaque mandatory<300..400>;
      /* length field is 2 bytes, cannot be empty */
uint16 longer<0..800>;
      /* zero to 400 16-bit unsigned integers */
```

3.5. Enumerateds

An additional sparse data type is available called enum or enumerated. Each definition is a different type. Only enumerateds of the same type may be assigned or compared. Every element of an enumerated must be assigned a value, as demonstrated in the following example. Since the elements of the enumerated are not ordered, they can be assigned any unique value, in any order.

```
enum { e1(v1), e2(v2), ... , en(vn) [[, (n)]] } Te;
```

Future extensions or additions to the protocol may define new values. Implementations need to be able to parse and ignore unknown values unless the definition of the field states otherwise.

An enumerated occupies as much space in the byte stream as would its maximal defined ordinal value. The following definition would cause one byte to be used to carry fields of type Color.

```
enum { red(3), blue(5), white(7) } Color;
```

One may optionally specify a value without its associated tag to force the width definition without defining a superfluous element.

In the following example, Taste will consume two bytes in the data stream but can only assume the values 1, 2, or 4 in the current version of the protocol.

```
enum { sweet(1), sour(2), bitter(4), (32000) } Taste;
```

The names of the elements of an enumeration are scoped within the defined type. In the first example, a fully qualified reference to the second element of the enumeration would be Color.blue. Such qualification is not required if the target of the assignment is well specified.

```
Color color = Color.blue;     /* overspecified, legal */
Color color = blue;           /* correct, type implicit */
```

The names assigned to enumerateds do not need to be unique. The numerical value can describe a range over which the same name applies. The value includes the minimum and maximum inclusive values in that range, separated by two period characters. This is principally useful for reserving regions of the space.

```
enum { sad(0), meh(1..254), happy(255) } Mood;
```

3.6. Constructed Types

Structure types may be constructed from primitive types for convenience. Each specification declares a new, unique type. The syntax for definition is much like that of C.

```
struct {
    T1 f1;
    T2 f2;
    ...
    Tn fn;
} T;
```

Fixed- and variable-length vector fields are allowed using the standard vector syntax. Structures V1 and V2 in the variants example below demonstrate this.

The fields within a structure may be qualified using the type's name, with a syntax much like that available for enumerateds. For example, T.f2 refers to the second field of the previous declaration.

3.7. Constants

Fields and variables may be assigned a fixed value using "=", as in:

```
struct {
    T1 f1 = 8;  /* T.f1 must always be 8 */
    T2 f2;
} T;
```

3.8. Variants

Defined structures may have variants based on some knowledge that is available within the environment. The selector must be an enumerated type that defines the possible variants the structure defines. Each arm of the select specifies the type of that variant's field and an optional field label. The mechanism by which the variant is selected at runtime is not prescribed by the presentation language.

```
struct {
    T1 f1;
    T2 f2;
    ....
    Tn fn;
    select (E) {
        case e1: Te1 [[fe1]];
        case e2: Te2 [[fe2]];
        ....
        case en: Ten [[fen]];
    };
} Tv;
```

For example:

```
enum { apple(0), orange(1) } VariantTag;

struct {
    uint16 number;
    opaque string<0..10>; /* variable length */
} V1;

struct {
    uint32 number;
    opaque string[10];    /* fixed length */
} V2;

struct {
    VariantTag type;
    select (VariantRecord.type) {
        case apple:  V1;
        case orange: V2;
    };
} VariantRecord;
```

4. Handshake Protocol

The handshake protocol is used to negotiate the security parameters of a connection. Handshake messages are supplied to the TLS record layer, where they are encapsulated within one or more TLSPlaintext or TLSCiphertext structures, which are processed and transmitted as specified by the current active connection state.

```
enum {
    client_hello(1),
    server_hello(2),
    new_session_ticket(4),
    end_of_early_data(5),
    encrypted_extensions(8),
    certificate(11),
    certificate_request(13),
    certificate_verify(15),
    finished(20),
    key_update(24),
    message_hash(254),
    (255)
} HandshakeType;

struct {
    HandshakeType msg_type;    /* handshake type */
    uint24 length;             /* bytes in message */
    select (Handshake.msg_type) {
        case client_hello:          ClientHello;
        case server_hello:          ServerHello;
        case end_of_early_data:     EndOfEarlyData;
        case encrypted_extensions:  EncryptedExtensions;
        case certificate_request:   CertificateRequest;
        case certificate:           Certificate;
        case certificate_verify:    CertificateVerify;
        case finished:              Finished;
        case new_session_ticket:    NewSessionTicket;
        case key_update:            KeyUpdate;
    };
} Handshake;
```

Protocol messages MUST be sent in the order defined in Section 4.4.1 and shown in the diagrams in Section 2. A peer which receives a handshake message in an unexpected order MUST abort the handshake with an "unexpected_message" alert.

New handshake message types are assigned by IANA as described in Section 11.

4.1. Key Exchange Messages

The key exchange messages are used to determine the security capabilities of the client and the server and to establish shared secrets including the traffic keys used to protect the rest of the handshake and the data.

4.1.1. Cryptographic Negotiation

In TLS, the cryptographic negotiation proceeds by the client offering the following four sets of options in its ClientHello:

- A list of cipher suites which indicates the AEAD algorithm/HKDF hash pairs which the client supports.

- A "supported_groups" (Section 4.2.7) extension which indicates the (EC)DHE groups which the client supports and a "key_share" (Section 4.2.8) extension which contains (EC)DHE shares for some or all of these groups.

- A "signature_algorithms" (Section 4.2.3) extension which indicates the signature algorithms which the client can accept.

- A "pre_shared_key" (Section 4.2.11) extension which contains a list of symmetric key identities known to the client and a "psk_key_exchange_modes" (Section 4.2.9) extension which indicates the key exchange modes that may be used with PSKs.

If the server does not select a PSK, then the first three of these options are entirely orthogonal: the server independently selects a cipher suite, an (EC)DHE group and key share for key establishment, and a signature algorithm/certificate pair to authenticate itself to the client. If there is no overlap between the received "supported_groups" and the groups supported by the server then the server MUST abort the handshake with a "handshake_failure" or an "insufficient_security" alert.

If the server selects a PSK, then it MUST also select a key establishment mode from the set indicated by client's "psk_key_exchange_modes" extension (at present, PSK alone or with (EC)DHE). Note that if the PSK can be used without (EC)DHE then non-overlap in the "supported_groups" parameters need not be fatal, as it is in the non-PSK case discussed in the previous paragraph.

If the server selects an (EC)DHE group and the client did not offer a compatible "key_share" extension in the initial ClientHello, the server MUST respond with a HelloRetryRequest (Section 4.1.4) message.

If the server successfully selects parameters and does not require a HelloRetryRequest, it indicates the selected parameters in the ServerHello as follows:

- If PSK is being used, then the server will send a "pre_shared_key" extension indicating the selected key.

- If PSK is not being used, then (EC)DHE and certificate-based authentication are always used.

- When (EC)DHE is in use, the server will also provide a "key_share" extension.

- When authenticating via a certificate, the server will send the Certificate (Section 4.4.2) and CertificateVerify (Section 4.4.3) messages. In TLS 1.3 as defined by this document, either a PSK or a certificate is always used, but not both. Future documents may define how to use them together.

If the server is unable to negotiate a supported set of parameters (i.e., there is no overlap between the client and server parameters), it MUST abort the handshake with either a "handshake_failure" or "insufficient_security" fatal alert (see Section 6).

4.1.2. Client Hello

When a client first connects to a server, it is REQUIRED to send the ClientHello as its first TLS message. The client will also send a ClientHello when the server has responded to its ClientHello with a HelloRetryRequest. In that case, the client MUST send the same ClientHello without modification, except:

- If a "key_share" extension was supplied in the HelloRetryRequest, replacing the list of shares with a list containing a single KeyShareEntry from the indicated group.

- Removing the "early_data" extension (Section 4.2.10) if one was present. Early data is not permitted after HelloRetryRequest.

- Including a "cookie" extension if one was provided in the HelloRetryRequest.

- Updating the "pre_shared_key" extension if present by recomputing the "obfuscated_ticket_age" and binder values and (optionally) removing any PSKs which are incompatible with the server’s indicated cipher suite.

- Optionally adding, removing, or changing the length of the "padding" extension [RFC7685].

- Other modifications that may be allowed by an extension defined in the future and present in the HelloRetryRequest.

Because TLS 1.3 forbids renegotiation, if a server has negotiated TLS 1.3 and receives a ClientHello at any other time, it MUST terminate the connection with an "unexpected_message" alert.

If a server established a TLS connection with a previous version of TLS and receives a TLS 1.3 ClientHello in a renegotiation, it MUST retain the previous protocol version. In particular, it MUST NOT negotiate TLS 1.3.

Structure of this message:

```
uint16 ProtocolVersion;
opaque Random[32];

uint8 CipherSuite[2];    /* Cryptographic suite selector */

struct {
    ProtocolVersion legacy_version = 0x0303;    /* TLS v1.2 */
    Random random;
    opaque legacy_session_id<0..32>;
    CipherSuite cipher_suites<2..2^16-2>;
    opaque legacy_compression_methods<1..2^8-1>;
    Extension extensions<8..2^16-1>;
} ClientHello;
```

legacy_version In previous versions of TLS, this field was used for version negotiation and represented the highest version number supported by the client. Experience has shown that many servers do not properly implement version negotiation, leading to "version intolerance" in which the server rejects an otherwise acceptable ClientHello with a version number higher than it supports. In TLS 1.3, the client indicates its version preferences in the "supported_versions" extension (Section 4.2.1) and the legacy_version field MUST be set to 0x0303, which is the version number for TLS 1.2. (See Appendix D for details about backward compatibility.)

random 32 bytes generated by a secure random number generator. See Appendix C for additional information.

legacy_session_id Versions of TLS before TLS 1.3 supported a "session resumption" feature which has been merged with Pre-Shared

Keys in this version (see Section 2.2). A client which has a cached session ID set by a pre-TLS 1.3 server SHOULD set this field to that value. In compatibility mode (see Appendix D.4) this field MUST be non-empty, so a client not offering a pre-TLS 1.3 session MUST generate a new 32-byte value. This value need not be random but SHOULD be unpredictable to avoid implementations fixating on a specific value (also known as ossification). Otherwise, it MUST be set as a zero length vector (i.e., a single zero byte length field).

cipher_suites This is a list of the symmetric cipher options supported by the client, specifically the record protection algorithm (including secret key length) and a hash to be used with HKDF, in descending order of client preference. If the list contains cipher suites that the server does not recognize, support or wish to use, the server MUST ignore those cipher suites and process the remaining ones as usual. Values are defined in Appendix B.4. If the client is attempting a PSK key establishment, it SHOULD advertise at least one cipher suite indicating a Hash associated with the PSK.

legacy_compression_methods Versions of TLS before 1.3 supported compression with the list of supported compression methods being sent in this field. For every TLS 1.3 ClientHello, this vector MUST contain exactly one byte, set to zero, which corresponds to the "null" compression method in prior versions of TLS. If a TLS 1.3 ClientHello is received with any other value in this field, the server MUST abort the handshake with an "illegal_parameter" alert. Note that TLS 1.3 servers might receive TLS 1.2 or prior ClientHellos which contain other compression methods and (if negotiating such a prior version) MUST follow the procedures for the appropriate prior version of TLS. TLS 1.3 ClientHellos are identified as having a legacy_version of 0x0303 and a supported_versions extension present with 0x0304 as the highest version indicated therein.

extensions Clients request extended functionality from servers by sending data in the extensions field. The actual "Extension" format is defined in Section 4.2. In TLS 1.3, use of certain extensions is mandatory, as functionality is moved into extensions to preserve ClientHello compatibility with previous versions of TLS. Servers MUST ignore unrecognized extensions.

All versions of TLS allow an extensions field to optionally follow the compression_methods field. TLS 1.3 ClientHello messages always contain extensions (minimally "supported_versions", otherwise they will be interpreted as TLS 1.2 ClientHello messages). However, TLS 1.3 servers might receive ClientHello messages without an extensions field from prior versions of TLS. The presence of extensions can be detected by determining whether there are bytes following the compression_methods field at the end of the ClientHello. Note that this method of detecting optional data differs from the normal TLS method of having a variable-length field, but it is used for compatibility with TLS before extensions were defined. TLS 1.3 servers will need to perform this check first and only attempt to negotiate TLS 1.3 if the "supported_versions" extension is present. If negotiating a version of TLS prior to 1.3, a server MUST check that the message either contains no data after legacy_compression_methods or that it contains a valid extensions block with no data following. If not, then it MUST abort the handshake with a "decode_error" alert.

In the event that a client requests additional functionality using extensions, and this functionality is not supplied by the server, the client MAY abort the handshake.

After sending the ClientHello message, the client waits for a ServerHello or HelloRetryRequest message. If early data is in use, the client may transmit early application data (Section 2.3) while waiting for the next handshake message.

4.1.3. Server Hello

The server will send this message in response to a ClientHello message to proceed with the handshake if it is able to negotiate an acceptable set of handshake parameters based on the ClientHello.

Structure of this message:

```
struct {
    ProtocolVersion legacy_version = 0x0303;    /* TLS v1.2 */
    Random random;
    opaque legacy_session_id_echo<0..32>;
    CipherSuite cipher_suite;
    uint8 legacy_compression_method = 0;
    Extension extensions<6..2^16-1>;
} ServerHello;
```

legacy_version In previous versions of TLS, this field was used for version negotiation and represented the selected version number for the connection. Unfortunately, some middleboxes fail when presented with new values. In TLS 1.3, the TLS server indicates its version using the "supported_versions" extension (Section 4.2.1), and the legacy_version field MUST be set to 0x0303, which is the version number for TLS 1.2. (See Appendix D for details about backward compatibility.)

random 32 bytes generated by a secure random number generator. See Appendix C for additional information. The last eight bytes MUST be overwritten as described below if negotiating TLS 1.2 or TLS 1.1, but the remaining bytes MUST be random. This structure is generated by the server and MUST be generated independently of the ClientHello.random.

legacy_session_id_echo The contents of the client's legacy_session_id field. Note that this field is echoed even if the client's value corresponded to a cached pre-TLS 1.3 session which the server has chosen not to resume. A client which receives a legacy_session_id_echo field that does not match what it sent in the ClientHello MUST abort the handshake with an "illegal_parameter" alert.

cipher_suite The single cipher suite selected by the server from the list in ClientHello.cipher_suites. A client which receives a cipher suite that was not offered MUST abort the handshake with an "illegal_parameter" alert.

legacy_compression_method A single byte which MUST have the value 0.

extensions A list of extensions. The ServerHello MUST only include extensions which are required to establish the cryptographic context and negotiate the protocol version. All TLS 1.3 ServerHello messages MUST contain the "supported_versions" extension. Current ServerHello messages additionally contain either the "pre_shared_key" or "key_share" extensions, or both when using a PSK with (EC)DHE key establishment. Other extensions are sent separately in the EncryptedExtensions message.

For reasons of backward compatibility with middleboxes (see Appendix D.4) the HelloRetryRequest message uses the same structure as the ServerHello, but with Random set to the special value of the SHA-256 of "HelloRetryRequest":

```
CF 21 AD 74 E5 9A 61 11 BE 1D 8C 02 1E 65 B8 91
C2 A2 11 16 7A BB 8C 5E 07 9E 09 E2 C8 A8 33 9C
```

Upon receiving a message with type server_hello, implementations MUST first examine the Random value and if it matches this value, process it as described in Section 4.1.4).

TLS 1.3 has a downgrade protection mechanism embedded in the server's random value. TLS 1.3 servers which negotiate TLS 1.2 or below in response to a ClientHello MUST set the last eight bytes of their Random value specially.

If negotiating TLS 1.2, TLS 1.3 servers MUST set the last eight bytes of their Random value to the bytes:

    44 4F 57 4E 47 52 44 01

If negotiating TLS 1.1 or below, TLS 1.3 servers MUST and TLS 1.2 servers SHOULD set the last eight bytes of their Random value to the bytes:

    44 4F 57 4E 47 52 44 00

TLS 1.3 clients receiving a ServerHello indicating TLS 1.2 or below MUST check that the last eight bytes are not equal to either of these values. TLS 1.2 clients SHOULD also check that the last eight bytes are not equal to the second value if the ServerHello indicates TLS 1.1 or below. If a match is found, the client MUST abort the handshake with an "illegal_parameter" alert. This mechanism provides limited protection against downgrade attacks over and above what is provided by the Finished exchange: because the ServerKeyExchange, a message present in TLS 1.2 and below, includes a signature over both random values, it is not possible for an active attacker to modify the random values without detection as long as ephemeral ciphers are used. It does not provide downgrade protection when static RSA is used.

Note: This is a change from [RFC5246], so in practice many TLS 1.2 clients and servers will not behave as specified above.

A legacy TLS client performing renegotiation with TLS 1.2 or prior and which receives a TLS 1.3 ServerHello during renegotiation MUST abort the handshake with a "protocol_version" alert. Note that renegotiation is not possible when TLS 1.3 has been negotiated.

RFC EDITOR: PLEASE REMOVE THE FOLLOWING PARAGRAPH Implementations of draft versions (see Section 4.2.1.1) of this specification SHOULD NOT implement this mechanism on either client and server. A pre-RFC client connecting to RFC servers, or vice versa, will appear to downgrade to TLS 1.2. With the mechanism enabled, this will cause an interoperability failure.

4.1.4. Hello Retry Request

The server will send this message in response to a ClientHello message if it is able to find an acceptable set of parameters but the ClientHello does not contain sufficient information to proceed with the handshake. As discussed in Section 4.1.3, the HelloRetryRequest has the same format as a ServerHello message, and the legacy_version, legacy_session_id_echo, cipher_suite, and legacy_compression methods fields have the same meaning. However, for convenience we discuss HelloRetryRequest throughout this document as if it were a distinct message.

The server's extensions MUST contain "supported_versions" and otherwise the server SHOULD send only the extensions necessary for the client to generate a correct ClientHello pair. As with ServerHello, a HelloRetryRequest MUST NOT contain any extensions that were not first offered by the client in its ClientHello, with the exception of optionally the "cookie" (see Section 4.2.2) extension.

Upon receipt of a HelloRetryRequest, the client MUST check the legacy_version, legacy_session_id_echo, cipher_suite, and legacy_compression_method as specified in Section 4.1.3 and then process the extensions, starting with determining the version using "supported_versions". Clients MUST abort the handshake with an "illegal_parameter" alert if the HelloRetryRequest would not result in any change in the ClientHello. If a client receives a second HelloRetryRequest in the same connection (i.e., where the ClientHello was itself in response to a HelloRetryRequest), it MUST abort the handshake with an "unexpected_message" alert.

Otherwise, the client MUST process all extensions in the HelloRetryRequest and send a second updated ClientHello. The HelloRetryRequest extensions defined in this specification are:

- supported_versions (see Section 4.2.1)

- cookie (see Section 4.2.2)

- key_share (see Section 4.2.8)

In addition, in its updated ClientHello, the client SHOULD NOT offer any pre-shared keys associated with a hash other than that of the selected cipher suite. This allows the client to avoid having to compute partial hash transcripts for multiple hashes in the second ClientHello. A client which receives a cipher suite that was not offered MUST abort the handshake. Servers MUST ensure that they negotiate the same cipher suite when receiving a conformant updated ClientHello (if the server selects the cipher suite as the first step in the negotiation, then this will happen automatically). Upon receiving the ServerHello, clients MUST check that the cipher suite supplied in the ServerHello is the same as that in the HelloRetryRequest and otherwise abort the handshake with an "illegal_parameter" alert.

The value of selected_version in the HelloRetryRequest "supported_versions" extension MUST be retained in the ServerHello, and a client MUST abort the handshake with an "illegal_parameter" alert if the value changes.

4.2. Extensions

A number of TLS messages contain tag-length-value encoded extensions structures.

```
struct {
    ExtensionType extension_type;
    opaque extension_data<0..2^16-1>;
} Extension;

enum {
    server_name(0),                             /* RFC 6066 */
    max_fragment_length(1),                     /* RFC 6066 */
    status_request(5),                          /* RFC 6066 */
    supported_groups(10),                       /* RFC 4492, 7919 */
    signature_algorithms(13),                   /* [[this document]] */
    use_srtp(14),                               /* RFC 5764 */
    heartbeat(15),                              /* RFC 6520 */
    application_layer_protocol_negotiation(16), /* RFC 7301 */
    signed_certificate_timestamp(18),           /* RFC 6962 */
    client_certificate_type(19),                /* RFC 7250 */
    server_certificate_type(20),                /* RFC 7250 */
    padding(21),                                /* RFC 7685 */
    pre_shared_key(41),                         /* [[this document]] */
    early_data(42),                             /* [[this document]] */
    supported_versions(43),                     /* [[this document]] */
    cookie(44),                                 /* [[this document]] */
    psk_key_exchange_modes(45),                 /* [[this document]] */
    certificate_authorities(47),                /* [[this document]] */
    oid_filters(48),                            /* [[this document]] */
    post_handshake_auth(49),                    /* [[this document]] */
    signature_algorithms_cert(50),              /* [[this document]] */
    key_share(51),                              /* [[this document]] */
    (65535)
} ExtensionType;
```

Here:

- "extension_type" identifies the particular extension type.

- "extension_data" contains information specific to the particular extension type.

The list of extension types is maintained by IANA as described in Section 11.

Extensions are generally structured in a request/response fashion, though some extensions are just indications with no corresponding response. The client sends its extension requests in the ClientHello message and the server sends its extension responses in the ServerHello, EncryptedExtensions, HelloRetryRequest and Certificate messages. The server sends extension requests in the CertificateRequest message which a client MAY respond to with a Certificate message. The server MAY also send unsolicited extensions in the NewSessionTicket, though the client does not respond directly to these.

Implementations MUST NOT send extension responses if the remote endpoint did not send the corresponding extension requests, with the exception of the "cookie" extension in HelloRetryRequest. Upon receiving such an extension, an endpoint MUST abort the handshake with an "unsupported_extension" alert.

The table below indicates the messages where a given extension may appear, using the following notation: CH (ClientHello), SH (ServerHello), EE (EncryptedExtensions), CT (Certificate), CR (CertificateRequest), NST (NewSessionTicket) and HRR (HelloRetryRequest). If an implementation receives an extension which it recognizes and which is not specified for the message in which it appears it MUST abort the handshake with an "illegal_parameter" alert.

Extension	TLS 1.3
server_name [RFC6066]	CH, EE
max_fragment_length [RFC6066]	CH, EE
status_request [RFC6066]	CH, CR, CT
supported_groups [RFC7919]	CH, EE
signature_algorithms [RFC5246]	CH, CR
use_srtp [RFC5764]	CH, EE
heartbeat [RFC6520]	CH, EE
application_layer_protocol_negotiation [RFC7301]	CH, EE
signed_certificate_timestamp [RFC6962]	CH, CR, CT
client_certificate_type [RFC7250]	CH, EE
server_certificate_type [RFC7250]	CH, EE
padding [RFC7685]	CH
key_share [[this document]]	CH, SH, HRR
pre_shared_key [[this document]]	CH, SH
psk_key_exchange_modes [[this document]]	CH
early_data [[this document]]	CH, EE, NST
cookie [[this document]]	CH, HRR
supported_versions [[this document]]	CH, SH, HRR
certificate_authorities [[this document]]	CH, CR
oid_filters [[this document]]	CR
post_handshake_auth [[this document]]	CH
signature_algorithms_cert [[this document]]	CH, CR

When multiple extensions of different types are present, the extensions MAY appear in any order, with the exception of "pre_shared_key" Section 4.2.11 which MUST be the last extension in the ClientHello. There MUST NOT be more than one extension of the same type in a given extension block.

In TLS 1.3, unlike TLS 1.2, extensions are negotiated for each handshake even when in resumption-PSK mode. However, 0-RTT parameters are those negotiated in the previous handshake; mismatches may require rejecting 0-RTT (see Section 4.2.10).

There are subtle (and not so subtle) interactions that may occur in this protocol between new features and existing features which may result in a significant reduction in overall security. The following considerations should be taken into account when designing new extensions:

- Some cases where a server does not agree to an extension are error conditions (e.g., the handshake cannot continue), and some are simply refusals to support particular features. In general, error alerts should be used for the former and a field in the server extension response for the latter.

- Extensions should, as far as possible, be designed to prevent any attack that forces use (or non-use) of a particular feature by manipulation of handshake messages. This principle should be followed regardless of whether the feature is believed to cause a security problem. Often the fact that the extension fields are included in the inputs to the Finished message hashes will be sufficient, but extreme care is needed when the extension changes the meaning of messages sent in the handshake phase. Designers and implementors should be aware of the fact that until the handshake has been authenticated, active attackers can modify messages and insert, remove, or replace extensions.

4.2.1. Supported Versions

```
struct {
    select (Handshake.msg_type) {
        case client_hello:
             ProtocolVersion versions<2..254>;

        case server_hello: /* and HelloRetryRequest */
             ProtocolVersion selected_version;
    };
} SupportedVersions;
```

The "supported_versions" extension is used by the client to indicate which versions of TLS it supports and by the server to indicate which version it is using. The extension contains a list of supported versions in preference order, with the most preferred version first. Implementations of this specification MUST send this extension in the ClientHello containing all versions of TLS which they are prepared to negotiate (for this specification, that means minimally 0x0304, but if previous versions of TLS are allowed to be negotiated, they MUST be present as well).

If this extension is not present, servers which are compliant with this specification, and which also support TLS 1.2, MUST negotiate TLS 1.2 or prior as specified in [RFC5246], even if ClientHello.legacy_version is 0x0304 or later. Servers MAY abort the handshake upon receiving a ClientHello with legacy_version 0x0304 or later.

If this extension is present in the ClientHello, servers MUST NOT use the ClientHello.legacy_version value for version negotiation and MUST use only the "supported_versions" extension to determine client preferences. Servers MUST only select a version of TLS present in that extension and MUST ignore any unknown versions that are present in that extension. Note that this mechanism makes it possible to negotiate a version prior to TLS 1.2 if one side supports a sparse range. Implementations of TLS 1.3 which choose to support prior versions of TLS SHOULD support TLS 1.2. Servers MUST be prepared to receive ClientHellos that include this extension but do not include 0x0304 in the list of versions.

A server which negotiates a version of TLS prior to TLS 1.3 MUST set ServerHello.version and MUST NOT send the "supported_versions" extension. A server which negotiates TLS 1.3 MUST respond by sending a "supported_versions" extension containing the selected version value (0x0304). It MUST set the ServerHello.legacy_version field to 0x0303 (TLS 1.2). Clients MUST check for this extension prior to processing the rest of the ServerHello (although they will have to parse the ServerHello in order to read the extension). If this extension is present, clients MUST ignore the ServerHello.legacy_version value and MUST use only the "supported_versions" extension to determine the selected version. If the "supported_versions" extension in the ServerHello contains a version not offered by the client or contains a version prior to TLS 1.3, the client MUST abort the handshake with an "illegal_parameter" alert.

4.2.1.1. Draft Version Indicator

RFC EDITOR: PLEASE REMOVE THIS SECTION

While the eventual version indicator for the RFC version of TLS 1.3 will be 0x0304, implementations of draft versions of this specification SHOULD instead advertise 0x7f00 | draft_version in the ServerHello and HelloRetryRequest "supported_versions" extension. For instance, draft-17 would be encoded as the 0x7f11. This allows pre-RFC implementations to safely negotiate with each other, even if they would otherwise be incompatible.

4.2.2. Cookie

```
struct {
    opaque cookie<1..2^16-1>;
} Cookie;
```

Cookies serve two primary purposes:

- Allowing the server to force the client to demonstrate reachability at their apparent network address (thus providing a measure of DoS protection). This is primarily useful for non-connection-oriented transports (see [RFC6347] for an example of this).

- Allowing the server to offload state to the client, thus allowing it to send a HelloRetryRequest without storing any state. The server can do this by storing the hash of the ClientHello in the HelloRetryRequest cookie (protected with some suitable integrity algorithm).

When sending a HelloRetryRequest, the server MAY provide a "cookie" extension to the client (this is an exception to the usual rule that the only extensions that may be sent are those that appear in the ClientHello). When sending the new ClientHello, the client MUST copy the contents of the extension received in the HelloRetryRequest into a "cookie" extension in the new ClientHello. Clients MUST NOT use cookies in their initial ClientHello in subsequent connections.

When a server is operating statelessly it may receive an unprotected record of type change_cipher_spec between the first and second ClientHello (see Section 5). Since the server is not storing any state this will appear as if it were the first message to be received. Servers operating statelessly MUST ignore these records.

4.2.3. Signature Algorithms

TLS 1.3 provides two extensions for indicating which signature algorithms may be used in digital signatures. The "signature_algorithms_cert" extension applies to signatures in certificates and the "signature_algorithms" extension, which originally appeared in TLS 1.2, applies to signatures in CertificateVerify messages. The keys found in certificates MUST also be of appropriate type for the signature algorithms they are used with. This is a particular issue for RSA keys and PSS signatures, as described below. If no "signature_algorithms_cert" extension is present, then the "signature_algorithms" extension also applies to signatures appearing in certificates. Clients which desire the server to authenticate itself via a certificate MUST send "signature_algorithms". If a server is authenticating via a certificate and the client has not sent a "signature_algorithms" extension, then the server MUST abort the handshake with a "missing_extension" alert (see Section 9.2).

The "signature_algorithms_cert" extension was added to allow implementations which supported different sets of algorithms for certificates and in TLS itself to clearly signal their capabilities. TLS 1.2 implementations SHOULD also process this extension. Implementations which have the same policy in both cases MAY omit the "signature_algorithms_cert" extension.

The "extension_data" field of these extensions contains a SignatureSchemeList value:

```
   enum {
       /* RSASSA-PKCS1-v1_5 algorithms */
       rsa_pkcs1_sha256(0x0401),
       rsa_pkcs1_sha384(0x0501),
       rsa_pkcs1_sha512(0x0601),

       /* ECDSA algorithms */
       ecdsa_secp256r1_sha256(0x0403),
       ecdsa_secp384r1_sha384(0x0503),
       ecdsa_secp521r1_sha512(0x0603),

       /* RSASSA-PSS algorithms with public key OID rsaEncryption */
       rsa_pss_rsae_sha256(0x0804),
       rsa_pss_rsae_sha384(0x0805),
       rsa_pss_rsae_sha512(0x0806),

       /* EdDSA algorithms */
       ed25519(0x0807),
       ed448(0x0808),

       /* RSASSA-PSS algorithms with public key OID RSASSA-PSS */
       rsa_pss_pss_sha256(0x0809),
       rsa_pss_pss_sha384(0x080a),
       rsa_pss_pss_sha512(0x080b),

       /* Legacy algorithms */
       rsa_pkcs1_sha1(0x0201),
       ecdsa_sha1(0x0203),

       /* Reserved Code Points */
       private_use(0xFE00..0xFFFF),
       (0xFFFF)
   } SignatureScheme;

   struct {
       SignatureScheme supported_signature_algorithms<2..2^16-2>;
   } SignatureSchemeList;
```

Note: This enum is named "SignatureScheme" because there is already a "SignatureAlgorithm" type in TLS 1.2, which this replaces. We use the term "signature algorithm" throughout the text.

Each SignatureScheme value lists a single signature algorithm that the client is willing to verify. The values are indicated in descending order of preference. Note that a signature algorithm takes as input an arbitrary-length message, rather than a digest. Algorithms which traditionally act on a digest should be defined in TLS to first hash the input with a specified hash algorithm and then proceed as usual. The code point groups listed above have the following meanings:

RSASSA-PKCS1-v1_5 algorithms Indicates a signature algorithm using RSASSA-PKCS1-v1_5 [RFC8017] with the corresponding hash algorithm as defined in [SHS]. These values refer solely to signatures which appear in certificates (see Section 4.4.2.2) and are not defined for use in signed TLS handshake messages, although they MAY appear in "signature_algorithms" and "signature_algorithms_cert" for backward compatibility with TLS 1.2,

ECDSA algorithms Indicates a signature algorithm using ECDSA [ECDSA], the corresponding curve as defined in ANSI X9.62 [X962] and FIPS 186-4 [DSS], and the corresponding hash algorithm as defined in [SHS]. The signature is represented as a DER-encoded [X690] ECDSA-Sig-Value structure.

RSASSA-PSS RSAE algorithms Indicates a signature algorithm using RSASSA-PSS [RFC8017] with mask generation function 1. The digest used in the mask generation function and the digest being signed are both the corresponding hash algorithm as defined in [SHS]. The length of the salt MUST be equal to the length of the output of the digest algorithm. If the public key is carried in an X.509 certificate, it MUST use the rsaEncryption OID [RFC5280].

EdDSA algorithms Indicates a signature algorithm using EdDSA as defined in [RFC8032] or its successors. Note that these correspond to the "PureEdDSA" algorithms and not the "prehash" variants.

RSASSA-PSS PSS algorithms Indicates a signature algorithm using RSASSA-PSS [RFC8017] with mask generation function 1. The digest used in the mask generation function and the digest being signed are both the corresponding hash algorithm as defined in [SHS]. The length of the salt MUST be equal to the length of the digest algorithm. If the public key is carried in an X.509 certificate, it MUST use the RSASSA-PSS OID [RFC5756]. When used in certificate signatures, the algorithm parameters MUST be DER encoded. If the corresponding public key’s parameters are present, then the parameters in the signature MUST be identical to those in the public key.

Legacy algorithms Indicates algorithms which are being deprecated because they use algorithms with known weaknesses, specifically SHA-1 which is used in this context with either with RSA using RSASSA-PKCS1-v1_5 or ECDSA. These values refer solely to signatures which appear in certificates (see Section 4.4.2.2) and are not defined for use in signed TLS handshake messages, although they MAY appear in "signature_algorithms" and "signature_algorithms_cert" for backward compatibility with TLS 1.2, Endpoints SHOULD NOT negotiate these algorithms but are permitted to do so solely for backward compatibility. Clients offering these values MUST list them as the lowest priority (listed after all other algorithms in SignatureSchemeList). TLS 1.3 servers MUST NOT offer a SHA-1 signed certificate unless no valid certificate chain can be produced without it (see Section 4.4.2.2).

The signatures on certificates that are self-signed or certificates that are trust anchors are not validated since they begin a certification path (see [RFC5280], Section 3.2). A certificate that begins a certification path MAY use a signature algorithm that is not advertised as being supported in the "signature_algorithms" extension.

Note that TLS 1.2 defines this extension differently. TLS 1.3 implementations willing to negotiate TLS 1.2 MUST behave in accordance with the requirements of [RFC5246] when negotiating that version. In particular:

- TLS 1.2 ClientHellos MAY omit this extension.

- In TLS 1.2, the extension contained hash/signature pairs. The pairs are encoded in two octets, so SignatureScheme values have been allocated to align with TLS 1.2's encoding. Some legacy pairs are left unallocated. These algorithms are deprecated as of TLS 1.3. They MUST NOT be offered or negotiated by any implementation. In particular, MD5 [SLOTH], SHA-224, and DSA MUST NOT be used.

- ECDSA signature schemes align with TLS 1.2's ECDSA hash/signature pairs. However, the old semantics did not constrain the signing curve. If TLS 1.2 is negotiated, implementations MUST be prepared to accept a signature that uses any curve that they advertised in the "supported_groups" extension.

- Implementations that advertise support for RSASSA-PSS (which is mandatory in TLS 1.3), MUST be prepared to accept a signature using that scheme even when TLS 1.2 is negotiated. In TLS 1.2, RSASSA-PSS is used with RSA cipher suites.

4.2.4. Certificate Authorities

The "certificate_authorities" extension is used to indicate the certificate authorities which an endpoint supports and which SHOULD be used by the receiving endpoint to guide certificate selection.

The body of the "certificate_authorities" extension consists of a CertificateAuthoritiesExtension structure.

```
   opaque DistinguishedName<1..2^16-1>;

   struct {
       DistinguishedName authorities<3..2^16-1>;
   } CertificateAuthoritiesExtension;
```

authorities A list of the distinguished names [X501] of acceptable certificate authorities, represented in DER-encoded [X690] format. These distinguished names specify a desired distinguished name for trust anchor or subordinate CA; thus, this message can be used to describe known trust anchors as well as a desired authorization space.

The client MAY send the "certificate_authorities" extension in the ClientHello message. The server MAY send it in the CertificateRequest message.

The "trusted_ca_keys" extension, which serves a similar purpose [RFC6066], but is more complicated, is not used in TLS 1.3 (although it may appear in ClientHello messages from clients which are offering prior versions of TLS).

4.2.5. OID Filters

The "oid_filters" extension allows servers to provide a set of OID/value pairs which it would like the client's certificate to match. This extension, if provided by the server, MUST only be sent in the CertificateRequest message.

```
   struct {
       opaque certificate_extension_oid<1..2^8-1>;
       opaque certificate_extension_values<0..2^16-1>;
   } OIDFilter;

   struct {
       OIDFilter filters<0..2^16-1>;
   } OIDFilterExtension;
```

filters A list of certificate extension OIDs [RFC5280] with their allowed value(s) and represented in DER-encoded [X690] format. Some certificate extension OIDs allow multiple values (e.g., Extended Key Usage). If the server has included a non-empty filters list, the client certificate included in the response MUST contain all of the specified extension OIDs that the client recognizes. For each extension OID recognized by the client, all of the specified values MUST be present in the client certificate (but the certificate MAY have other values as well). However, the client MUST ignore and skip any unrecognized certificate extension OIDs. If the client ignored some of the required certificate extension OIDs and supplied a certificate that does not satisfy the request, the server MAY at its discretion either continue the connection without client authentication, or abort the handshake with an "unsupported_certificate" alert. Any given OID MUST NOT appear more than once in the filters list.

PKIX RFCs define a variety of certificate extension OIDs and their corresponding value types. Depending on the type, matching certificate extension values are not necessarily bitwise-equal. It is expected that TLS implementations will rely on their PKI libraries to perform certificate selection using certificate extension OIDs.

This document defines matching rules for two standard certificate extensions defined in [RFC5280]:

- The Key Usage extension in a certificate matches the request when all key usage bits asserted in the request are also asserted in the Key Usage certificate extension.

- The Extended Key Usage extension in a certificate matches the request when all key purpose OIDs present in the request are also found in the Extended Key Usage certificate extension. The special anyExtendedKeyUsage OID MUST NOT be used in the request.

Separate specifications may define matching rules for other certificate extensions.

4.2.6. Post-Handshake Client Authentication

The "post_handshake_auth" extension is used to indicate that a client is willing to perform post-handshake authentication (Section 4.6.2). Servers MUST NOT send a post-handshake CertificateRequest to clients which do not offer this extension. Servers MUST NOT send this extension.

```
struct {} PostHandshakeAuth;
```

The "extension_data" field of the "post_handshake_auth" extension is zero length.

4.2.7. Negotiated Groups

When sent by the client, the "supported_groups" extension indicates the named groups which the client supports for key exchange, ordered from most preferred to least preferred.

Note: In versions of TLS prior to TLS 1.3, this extension was named "elliptic_curves" and only contained elliptic curve groups. See [RFC4492] and [RFC7919]. This extension was also used to negotiate ECDSA curves. Signature algorithms are now negotiated independently (see Section 4.2.3).

The "extension_data" field of this extension contains a "NamedGroupList" value:

```
enum {

    /* Elliptic Curve Groups (ECDHE) */
    secp256r1(0x0017), secp384r1(0x0018), secp521r1(0x0019),
    x25519(0x001D), x448(0x001E),

    /* Finite Field Groups (DHE) */
    ffdhe2048(0x0100), ffdhe3072(0x0101), ffdhe4096(0x0102),
    ffdhe6144(0x0103), ffdhe8192(0x0104),

    /* Reserved Code Points */
    ffdhe_private_use(0x01FC..0x01FF),
    ecdhe_private_use(0xFE00..0xFEFF),
    (0xFFFF)
} NamedGroup;

struct {
    NamedGroup named_group_list<2..2^16-1>;
} NamedGroupList;
```

Elliptic Curve Groups (ECDHE) Indicates support for the corresponding named curve, defined either in FIPS 186-4 [DSS] or in [RFC7748]. Values 0xFE00 through 0xFEFF are reserved for private use.

Finite Field Groups (DHE) Indicates support of the corresponding finite field group, defined in [RFC7919]. Values 0x01FC through 0x01FF are reserved for private use.

Items in named_group_list are ordered according to the client's preferences (most preferred choice first).

As of TLS 1.3, servers are permitted to send the "supported_groups" extension to the client. Clients MUST NOT act upon any information found in "supported_groups" prior to successful completion of the handshake but MAY use the information learned from a successfully completed handshake to change what groups they use in their "key_share" extension in subsequent connections. If the server has a group it prefers to the ones in the "key_share" extension but is still willing to accept the ClientHello, it SHOULD send "supported_groups" to update the client's view of its preferences; this extension SHOULD contain all groups the server supports, regardless of whether they are currently supported by the client.

4.2.8. Key Share

The "key_share" extension contains the endpoint's cryptographic parameters.

Clients MAY send an empty client_shares vector in order to request group selection from the server at the cost of an additional round trip. (see Section 4.1.4)

```
struct {
    NamedGroup group;
    opaque key_exchange<1..2^16-1>;
} KeyShareEntry;
```

group The named group for the key being exchanged.

key_exchange Key exchange information. The contents of this field are determined by the specified group and its corresponding definition. Finite Field Diffie-Hellman [DH] parameters are described in Section 4.2.8.1; Elliptic Curve Diffie-Hellman parameters are described in Section 4.2.8.2.

In the ClientHello message, the "extension_data" field of this extension contains a "KeyShareClientHello" value:

```
struct {
    KeyShareEntry client_shares<0..2^16-1>;
} KeyShareClientHello;
```

client_shares A list of offered KeyShareEntry values in descending order of client preference.

This vector MAY be empty if the client is requesting a HelloRetryRequest. Each KeyShareEntry value MUST correspond to a group offered in the "supported_groups" extension and MUST appear in the same order. However, the values MAY be a non-contiguous subset of the "supported_groups" extension and MAY omit the most preferred groups. Such a situation could arise if the most preferred groups are new and unlikely to be supported in enough places to make pregenerating key shares for them efficient.

Clients can offer as many KeyShareEntry values as the number of supported groups it is offering, each representing a single set of key exchange parameters. For instance, a client might offer shares for several elliptic curves or multiple FFDHE groups. The key_exchange values for each KeyShareEntry MUST be generated independently. Clients MUST NOT offer multiple KeyShareEntry values for the same group. Clients MUST NOT offer any KeyShareEntry values for groups not listed in the client's "supported_groups" extension. Servers MAY check for violations of these rules and abort the handshake with an "illegal_parameter" alert if one is violated.

In a HelloRetryRequest message, the "extension_data" field of this extension contains a KeyShareHelloRetryRequest value:

```
struct {
    NamedGroup selected_group;
} KeyShareHelloRetryRequest;
```

selected_group The mutually supported group the server intends to negotiate and is requesting a retried ClientHello/KeyShare for.

Upon receipt of this extension in a HelloRetryRequest, the client MUST verify that (1) the selected_group field corresponds to a group which was provided in the "supported_groups" extension in the original ClientHello; and (2) the selected_group field does not correspond to a group which was provided in the "key_share" extension in the original ClientHello. If either of these checks fails, then the client MUST abort the handshake with an "illegal_parameter" alert. Otherwise, when sending the new ClientHello, the client MUST replace the original "key_share" extension with one containing only a new KeyShareEntry for the group indicated in the selected_group field of the triggering HelloRetryRequest.

In a ServerHello message, the "extension_data" field of this extension contains a KeyShareServerHello value:

```
struct {
    KeyShareEntry server_share;
} KeyShareServerHello;
```

server_share A single KeyShareEntry value that is in the same group as one of the client's shares.

If using (EC)DHE key establishment, servers offer exactly one KeyShareEntry in the ServerHello. This value MUST be in the same group as the KeyShareEntry value offered by the client that the server has selected for the negotiated key exchange. Servers MUST NOT send a KeyShareEntry for any group not indicated in the "supported_groups" extension and MUST NOT send a KeyShareEntry when using the "psk_ke" PskKeyExchangeMode. If using (EC)DHE key establishment, and a HelloRetryRequest containing a "key_share" extension was received by the client, the client MUST verify that the selected NamedGroup in the ServerHello is the same as that in the HelloRetryRequest. If this check fails, the client MUST abort the handshake with an "illegal_parameter" alert.

4.2.8.1. Diffie-Hellman Parameters

Diffie-Hellman [DH] parameters for both clients and servers are encoded in the opaque key_exchange field of a KeyShareEntry in a KeyShare structure. The opaque value contains the Diffie-Hellman public value (Y = g^X mod p) for the specified group (see [RFC7919] for group definitions) encoded as a big-endian integer and padded to the left with zeros to the size of p in bytes.

Note: For a given Diffie-Hellman group, the padding results in all public keys having the same length.

Peers MUST validate each other's public key Y by ensuring that 1 < Y < p-1. This check ensures that the remote peer is properly behaved and isn't forcing the local system into a small subgroup.

4.2.8.2. ECDHE Parameters

ECDHE parameters for both clients and servers are encoded in the opaque key_exchange field of a KeyShareEntry in a KeyShare structure.

For secp256r1, secp384r1 and secp521r1, the contents are the serialized value of the following struct:

```
struct {
    uint8 legacy_form = 4;
    opaque X[coordinate_length];
    opaque Y[coordinate_length];
} UncompressedPointRepresentation;
```

X and Y respectively are the binary representations of the x and y values in network byte order. There are no internal length markers, so each number representation occupies as many octets as implied by the curve parameters. For P-256 this means that each of X and Y use 32 octets, padded on the left by zeros if necessary. For P-384 they take 48 octets each, and for P-521 they take 66 octets each.

For the curves secp256r1, secp384r1 and secp521r1, peers MUST validate each other's public value Q by ensuring that the point is a valid point on the elliptic curve. The appropriate validation procedures are defined in Section 4.3.7 of [X962] and alternatively in Section 5.6.2.3 of [KEYAGREEMENT]. This process consists of three steps: (1) verify that Q is not the point at infinity (O), (2) verify that for Q = (x, y) both integers x and y are in the correct interval, (3) ensure that (x, y) is a correct solution to the elliptic curve equation. For these curves, implementers do not need to verify membership in the correct subgroup.

For X25519 and X448, the contents of the public value are the byte string inputs and outputs of the corresponding functions defined in [RFC7748], 32 bytes for X25519 and 56 bytes for X448.

Note: Versions of TLS prior to 1.3 permitted point format negotiation; TLS 1.3 removes this feature in favor of a single point format for each curve.

4.2.9. Pre-Shared Key Exchange Modes

In order to use PSKs, clients MUST also send a "psk_key_exchange_modes" extension. The semantics of this extension are that the client only supports the use of PSKs with these modes, which restricts both the use of PSKs offered in this ClientHello and those which the server might supply via NewSessionTicket.

A client MUST provide a "psk_key_exchange_modes" extension if it offers a "pre_shared_key" extension. If clients offer "pre_shared_key" without a "psk_key_exchange_modes" extension, servers MUST abort the handshake. Servers MUST NOT select a key exchange mode that is not listed by the client. This extension also restricts the modes for use with PSK resumption; servers SHOULD NOT send NewSessionTicket with tickets that are not compatible with the advertised modes; however, if a server does so, the impact will just be that the client's attempts at resumption fail.

The server MUST NOT send a "psk_key_exchange_modes" extension.

```
enum { psk_ke(0), psk_dhe_ke(1), (255) } PskKeyExchangeMode;

struct {
    PskKeyExchangeMode ke_modes<1..255>;
} PskKeyExchangeModes;
```

psk_ke PSK-only key establishment. In this mode, the server MUST NOT supply a "key_share" value.

psk_dhe_ke PSK with (EC)DHE key establishment. In this mode, the client and server MUST supply "key_share" values as described in Section 4.2.8.

Any future values that are allocated must ensure that the transmitted protocol messages unambiguously identify which mode was selected by the server; at present, this is indicated by the presence of the "key_share" in the ServerHello.

4.2.10. Early Data Indication

When a PSK is used and early data is allowed for that PSK, the client can send application data in its first flight of messages. If the client opts to do so, it MUST supply both the "early_data" extension as well as the "pre_shared_key" extension.

The "extension_data" field of this extension contains an "EarlyDataIndication" value.

```
struct {} Empty;

struct {
    select (Handshake.msg_type) {
        case new_session_ticket:   uint32 max_early_data_size;
        case client_hello:         Empty;
        case encrypted_extensions: Empty;
    };
} EarlyDataIndication;
```

See Section 4.6.1 for the use of the max_early_data_size field.

The parameters for the 0-RTT data (version, symmetric cipher suite, ALPN protocol, etc.) are those associated with the PSK in use. For externally provisioned PSKs, the associated values are those provisioned along with the key. For PSKs established via a NewSessionTicket message, the associated values are those which were negotiated in the connection which established the PSK. The PSK used to encrypt the early data MUST be the first PSK listed in the client’s "pre_shared_key" extension.

For PSKs provisioned via NewSessionTicket, a server MUST validate that the ticket age for the selected PSK identity (computed by subtracting ticket_age_add from PskIdentity.obfuscated_ticket_age modulo 2^32) is within a small tolerance of the time since the ticket was issued (see Section 8). If it is not, the server SHOULD proceed with the handshake but reject 0-RTT, and SHOULD NOT take any other action that assumes that this ClientHello is fresh.

0-RTT messages sent in the first flight have the same (encrypted) content types as messages of the same type sent in other flights (handshake and application_data) but are protected under different keys. After receiving the server's Finished message, if the server has accepted early data, an EndOfEarlyData message will be sent to indicate the key change. This message will be encrypted with the 0-RTT traffic keys.

A server which receives an "early_data" extension MUST behave in one of three ways:

- Ignore the extension and return a regular 1-RTT response. The server then skips past early data by attempting to deprotect received records using the handshake traffic key, discarding records which fail deprotection (up to the configured max_early_data_size). Once a record is deprotected successfully, it is treated as the start of the client's second flight and the the server proceeds as with an ordinary 1-RTT handshake.

- Request that the client send another ClientHello by responding with a HelloRetryRequest. A client MUST NOT include the "early_data" extension in its followup ClientHello. The server then ignores early data by skipping all records with external content type of "application_data" (indicating that they are encrypted), up to the configured max_early_data_size.

- Return its own "early_data" extension in EncryptedExtensions, indicating that it intends to process the early data. It is not possible for the server to accept only a subset of the early data messages. Even though the server sends a message accepting early data, the actual early data itself may already be in flight by the time the server generates this message.

In order to accept early data, the server MUST have accepted a PSK cipher suite and selected the first key offered in the client's "pre_shared_key" extension. In addition, it MUST verify that the following values are the same as those associated with the selected PSK:

- The TLS version number

- The selected cipher suite

- The selected ALPN [RFC7301] protocol, if any

These requirements are a superset of those needed to perform a 1-RTT handshake using the PSK in question. For externally established PSKs, the associated values are those provisioned along with the key. For PSKs established via a NewSessionTicket message, the associated values are those negotiated in the connection during which the ticket was established.

Future extensions MUST define their interaction with 0-RTT.

If any of these checks fail, the server MUST NOT respond with the extension and must discard all the first flight data using one of the first two mechanisms listed above (thus falling back to 1-RTT or 2-RTT). If the client attempts a 0-RTT handshake but the server rejects it, the server will generally not have the 0-RTT record protection keys and must instead use trial decryption (either with the 1-RTT handshake keys or by looking for a cleartext ClientHello in the case of HelloRetryRequest) to find the first non-0-RTT message.

If the server chooses to accept the "early_data" extension, then it MUST comply with the same error handling requirements specified for all records when processing early data records. Specifically, if the server fails to decrypt a 0-RTT record following an accepted "early_data" extension it MUST terminate the connection with a "bad_record_mac" alert as per Section 5.2.

If the server rejects the "early_data" extension, the client application MAY opt to retransmit the application data previously sent in early data once the handshake has been completed. Note that automatic re-transmission of early data could result in assumptions about the status of the connection being incorrect. For instance, when the negotiated connection selects a different ALPN protocol from what was used for the early data, an application might need to construct different messages. Similarly, if early data assumes anything about the connection state, it might be sent in error after the handshake completes.

A TLS implementation SHOULD NOT automatically re-send early data; applications are in a better position to decide when re-transmission is appropriate. A TLS implementation MUST NOT automatically re-send early data unless the negotiated connection selects the same ALPN protocol.

4.2.11. Pre-Shared Key Extension

The "pre_shared_key" extension is used to negotiate the identity of the pre-shared key to be used with a given handshake in association with PSK key establishment.

The "extension_data" field of this extension contains a "PreSharedKeyExtension" value:

```
struct {
    opaque identity<1..2^16-1>;
    uint32 obfuscated_ticket_age;
} PskIdentity;

opaque PskBinderEntry<32..255>;

struct {
    PskIdentity identities<7..2^16-1>;
    PskBinderEntry binders<33..2^16-1>;
} OfferedPsks;

struct {
    select (Handshake.msg_type) {
        case client_hello: OfferedPsks;
        case server_hello: uint16 selected_identity;
    };
} PreSharedKeyExtension;
```

identity A label for a key. For instance, a ticket defined in Appendix B.3.4 or a label for a pre-shared key established externally.

obfuscated_ticket_age An obfuscated version of the age of the key. Section 4.2.11.1 describes how to form this value for identities established via the NewSessionTicket message. For identities established externally an obfuscated_ticket_age of 0 SHOULD be used, and servers MUST ignore the value.

identities A list of the identities that the client is willing to negotiate with the server. If sent alongside the "early_data" extension (see Section 4.2.10), the first identity is the one used for 0-RTT data.

binders A series of HMAC values, one for each PSK offered in the "pre_shared_keys" extension and in the same order, computed as described below.

selected_identity The server's chosen identity expressed as a (0-based) index into the identities in the client's list.

Each PSK is associated with a single Hash algorithm. For PSKs established via the ticket mechanism (Section 4.6.1), this is the KDF Hash algorithm on the connection where the ticket was established. For externally established PSKs, the Hash algorithm MUST be set when the PSK is established, or default to SHA-256 if no such algorithm is defined. The server MUST ensure that it selects a compatible PSK (if any) and cipher suite.

In TLS versions prior to TLS 1.3, the Server Name Identification (SNI) value was intended to be associated with the session (Section 3 of [RFC6066]), with the server being required to enforce that the SNI value associated with the session matches the one specified in the resumption handshake. However, in reality the implementations were not consistent on which of two supplied SNI values they would use, leading to the consistency requirement being de-facto enforced by the clients. In TLS 1.3, the SNI value is always explicitly specified in the resumption handshake, and there is no need for the server to associate an SNI value with the ticket. Clients, however, SHOULD store the SNI with the PSK to fulfill the requirements of Section 4.6.1.

Implementor's note: when session resumption is the primary use case of PSKs the most straightforward way to implement the PSK/cipher suite matching requirements is to negotiate the cipher suite first and then exclude any incompatible PSKs. Any unknown PSKs (e.g., they are not in the PSK database or are encrypted with an unknown key) SHOULD simply be ignored. If no acceptable PSKs are found, the server SHOULD perform a non-PSK handshake if possible. If backwards compatibility is important, client provided, externally established PSKs SHOULD influence cipher suite selection.

Prior to accepting PSK key establishment, the server MUST validate the corresponding binder value (see Section 4.2.11.2 below). If this value is not present or does not validate, the server MUST abort the handshake. Servers SHOULD NOT attempt to validate multiple binders; rather they SHOULD select a single PSK and validate solely the binder that corresponds to that PSK. See [Section 8.2] and [Appendix E.6] for the security rationale for this requirement. In order to accept PSK key establishment, the server sends a "pre_shared_key" extension indicating the selected identity.

Clients MUST verify that the server's selected_identity is within the range supplied by the client, that the server selected a cipher suite indicating a Hash associated with the PSK and that a server "key_share" extension is present if required by the ClientHello "psk_key_exchange_modes". If these values are not consistent the client MUST abort the handshake with an "illegal_parameter" alert.

If the server supplies an "early_data" extension, the client MUST verify that the server's selected_identity is 0. If any other value is returned, the client MUST abort the handshake with an "illegal_parameter" alert.

The "pre_shared_key" extension MUST be the last extension in the ClientHello (this facilitates implementation as described below). Servers MUST check that it is the last extension and otherwise fail the handshake with an "illegal_parameter" alert.

4.2.11.1. Ticket Age

The client's view of the age of a ticket is the time since the receipt of the NewSessionTicket message. Clients MUST NOT attempt to use tickets which have ages greater than the "ticket_lifetime" value which was provided with the ticket. The "obfuscated_ticket_age" field of each PskIdentity contains an obfuscated version of the ticket age formed by taking the age in milliseconds and adding the "ticket_age_add" value that was included with the ticket (see Section 4.6.1), modulo 2^32. This addition prevents passive observers from correlating connections unless tickets are reused. Note that the "ticket_lifetime" field in the NewSessionTicket message is in seconds but the "obfuscated_ticket_age" is in milliseconds. Because ticket lifetimes are restricted to a week, 32 bits is enough to represent any plausible age, even in milliseconds.

4.2.11.2. PSK Binder

The PSK binder value forms a binding between a PSK and the current handshake, as well as a binding between the handshake in which the PSK was generated (if via a NewSessionTicket message) and the current handshake. Each entry in the binders list is computed as an HMAC over a transcript hash (see Section 4.4.1) containing a partial ClientHello up to and including the PreSharedKeyExtension.identities field. That is, it includes all of the ClientHello but not the binders list itself. The length fields for the message (including the overall length, the length of the extensions block, and the length of the "pre_shared_key" extension) are all set as if binders of the correct lengths were present.

The PskBinderEntry is computed in the same way as the Finished message (Section 4.4.4) but with the BaseKey being the binder_key derived via the key schedule from the corresponding PSK which is being offered (see Section 7.1).

If the handshake includes a HelloRetryRequest, the initial ClientHello and HelloRetryRequest are included in the transcript along with the new ClientHello. For instance, if the client sends ClientHello1, its binder will be computed over:

```
Transcript-Hash(Truncate(ClientHello1))
```

Where Truncate() removes the binders list from the ClientHello.

If the server responds with HelloRetryRequest, and the client then sends ClientHello2, its binder will be computed over:

```
Transcript-Hash(ClientHello1,
                HelloRetryRequest,
                Truncate(ClientHello2))
```

The full ClientHello1/ClientHello2 is included in all other handshake hash computations. Note that in the first flight, Truncate(ClientHello1) is hashed directly, but in the second flight, ClientHello1 is hashed and then reinjected as a "message_hash" message, as described in Section 4.4.1.

4.2.11.3. Processing Order

Clients are permitted to "stream" 0-RTT data until they receive the server’s Finished, only then sending the EndOfEarlyData message, followed by the rest of the handshake. In order to avoid deadlocks, when accepting "early_data", servers MUST process the client’s ClientHello and then immediately send their flight of messages, rather than waiting for the client’s EndOfEarlyData message before sending its ServerHello.

4.3. Server Parameters

The next two messages from the server, EncryptedExtensions and CertificateRequest, contain information from the server that determines the rest of the handshake. These messages are encrypted with keys derived from the server_handshake_traffic_secret.

4.3.1. Encrypted Extensions

In all handshakes, the server MUST send the EncryptedExtensions message immediately after the ServerHello message. This is the first message that is encrypted under keys derived from the server_handshake_traffic_secret.

The EncryptedExtensions message contains extensions that can be protected, i.e., any which are not needed to establish the cryptographic context, but which are not associated with individual certificates. The client MUST check EncryptedExtensions for the presence of any forbidden extensions and if any are found MUST abort the handshake with an "illegal_parameter" alert.

Structure of this message:

```
struct {
    Extension extensions<0..2^16-1>;
} EncryptedExtensions;
```

extensions A list of extensions. For more information, see the table in Section 4.2.

4.3.2. Certificate Request

A server which is authenticating with a certificate MAY optionally request a certificate from the client. This message, if sent, MUST follow EncryptedExtensions.

Structure of this message:

```
struct {
    opaque certificate_request_context<0..2^8-1>;
    Extension extensions<2..2^16-1>;
} CertificateRequest;
```

certificate_request_context An opaque string which identifies the certificate request and which will be echoed in the client's Certificate message. The certificate_request_context MUST be unique within the scope of this connection (thus preventing replay of client CertificateVerify messages). This field SHALL be zero length unless used for the post-handshake authentication exchanges described in Section 4.6.2. When requesting post-handshake authentication, the server SHOULD make the context unpredictable to the client (e.g., by randomly generating it) in order to prevent an attacker who has temporary access to the client's private key from pre-computing valid CertificateVerify messages.

extensions A set of extensions describing the parameters of the certificate being requested. The "signature_algorithms" extension MUST be specified, and other extensions may optionally be included if defined for this message. Clients MUST ignore unrecognized extensions.

In prior versions of TLS, the CertificateRequest message carried a list of signature algorithms and certificate authorities which the server would accept. In TLS 1.3 the former is expressed by sending the "signature_algorithms" and optionally "signature_algorithms_cert" extensions. The latter is expressed by sending the "certificate_authorities" extension (see Section 4.2.4).

Servers which are authenticating with a PSK MUST NOT send the CertificateRequest message in the main handshake, though they MAY send it in post-handshake authentication (see Section 4.6.2) provided that the client has sent the "post_handshake_auth" extension (see Section 4.2.6).

4.4. Authentication Messages

As discussed in Section 2, TLS generally uses a common set of messages for authentication, key confirmation, and handshake integrity: Certificate, CertificateVerify, and Finished. (The PreSharedKey binders also perform key confirmation, in a similar fashion.) These three messages are always sent as the last messages in their handshake flight. The Certificate and CertificateVerify messages are only sent under certain circumstances, as defined below. The Finished message is always sent as part of the Authentication block. These messages are encrypted under keys derived from [sender]_handshake_traffic_secret.

The computations for the Authentication messages all uniformly take the following inputs:

- The certificate and signing key to be used.

- A Handshake Context consisting of the set of messages to be included in the transcript hash.

- A base key to be used to compute a MAC key.

Based on these inputs, the messages then contain:

Certificate The certificate to be used for authentication, and any supporting certificates in the chain. Note that certificate-based client authentication is not available in PSK (including 0-RTT) flows.

CertificateVerify A signature over the value Transcript-Hash(Handshake Context, Certificate)

Finished A MAC over the value Transcript-Hash(Handshake Context, Certificate, CertificateVerify) using a MAC key derived from the base key.

The following table defines the Handshake Context and MAC Base Key for each scenario:

Mode	Handshake Context	Base Key
Server	ClientHello ... later of EncryptedExtensions/CertificateRequest	server_handshake_traffic_secret
Client	ClientHello ... later of server Finished/EndOfEarlyData	client_handshake_traffic_secret
Post-Handshake	ClientHello ... client Finished + CertificateRequest	client_application_traffic_secret_N

4.4.1. The Transcript Hash

Many of the cryptographic computations in TLS make use of a transcript hash. This value is computed by hashing the concatenation of each included handshake message, including the handshake message header carrying the handshake message type and length fields, but not including record layer headers. I.e.,

```
 Transcript-Hash(M1, M2, ... Mn) = Hash(M1 || M2 || ... || Mn)
```

As an exception to this general rule, when the server responds to a ClientHello with a HelloRetryRequest, the value of ClientHello1 is replaced with a special synthetic handshake message of handshake type "message_hash" containing Hash(ClientHello1). I.e.,

```
Transcript-Hash(ClientHello1, HelloRetryRequest, ... Mn) =
    Hash(message_hash ||        /* Handshake type */
         00 00 Hash.length ||   /* Handshake message length (bytes) */
         Hash(ClientHello1) ||  /* Hash of ClientHello1 */
         HelloRetryRequest || ... || Mn)
```

The reason for this construction is to allow the server to do a stateless HelloRetryRequest by storing just the hash of ClientHello1 in the cookie, rather than requiring it to export the entire intermediate hash state (see Section 4.2.2).

For concreteness, the transcript hash is always taken from the following sequence of handshake messages, starting at the first ClientHello and including only those messages that were sent:

ClientHello, HelloRetryRequest, ClientHello, ServerHello, EncryptedExtensions, server CertificateRequest, server Certificate, server CertificateVerify, server Finished, EndOfEarlyData, client Certificate, client CertificateVerify, client Finished.

In general, implementations can implement the transcript by keeping a running transcript hash value based on the negotiated hash. Note, however, that subsequent post-handshake authentications do not include each other, just the messages through the end of the main handshake.

4.4.2. Certificate

This message conveys the endpoint’s certificate chain to the peer.

The server MUST send a Certificate message whenever the agreed-upon key exchange method uses certificates for authentication (this includes all key exchange methods defined in this document except PSK).

The client MUST send a Certificate message if and only if the server has requested client authentication via a CertificateRequest message (Section 4.3.2). If the server requests client authentication but no suitable certificate is available, the client MUST send a Certificate message containing no certificates (i.e., with the "certificate_list" field having length 0). A Finished message MUST be sent regardless of whether the Certificate message is empty.

Structure of this message:

```
/* Managed by IANA */
enum {
    X509(0),
    RawPublicKey(2),
    (255)
} CertificateType;

struct {
    select (certificate_type) {
        case RawPublicKey:
          /* From RFC 7250 ASN.1_subjectPublicKeyInfo */
          opaque ASN1_subjectPublicKeyInfo<1..2^24-1>;

        case X509:
          opaque cert_data<1..2^24-1>;
    };
    Extension extensions<0..2^16-1>;
} CertificateEntry;

struct {
    opaque certificate_request_context<0..2^8-1>;
    CertificateEntry certificate_list<0..2^24-1>;
} Certificate;
```

certificate_request_context If this message is in response to a CertificateRequest, the value of certificate_request_context in that message. Otherwise (in the case of server authentication), this field SHALL be zero length.

certificate_list This is a sequence (chain) of CertificateEntry structures, each containing a single certificate and set of extensions.

extensions: A set of extension values for the CertificateEntry. The "Extension" format is defined in Section 4.2. Valid extensions for server certificates at present include OCSP Status extension ([RFC6066]) and SignedCertificateTimestamps ([RFC6962]); future extensions may be defined for this message as well. Extensions in the Certificate message from the server MUST correspond to ones from the ClientHello message. Extensions in the Certificate from the client MUST correspond with extensions in the CertificateRequest message from the server. If an extension applies to the entire chain, it SHOULD be included in the first CertificateEntry.

If the corresponding certificate type extension ("server_certificate_type" or "client_certificate_type") was not negotiated in Encrypted Extensions, or the X.509 certificate type was negotiated, then each CertificateEntry contains a DER-encoded X.509 certificate. The sender's certificate MUST come in the first CertificateEntry in the list. Each following certificate SHOULD directly certify the one immediately preceding it. Because certificate validation requires that trust anchors be distributed independently, a certificate that specifies a trust anchor MAY be omitted from the chain, provided that supported peers are known to possess any omitted certificates.

Note: Prior to TLS 1.3, "certificate_list" ordering required each certificate to certify the one immediately preceding it; however, some implementations allowed some flexibility. Servers sometimes send both a current and deprecated intermediate for transitional purposes, and others are simply configured incorrectly, but these cases can nonetheless be validated properly. For maximum compatibility, all implementations SHOULD be prepared to handle potentially extraneous certificates and arbitrary orderings from any TLS version, with the exception of the end-entity certificate which MUST be first.

If the RawPublicKey certificate type was negotiated, then the certificate_list MUST contain no more than one CertificateEntry, which contains an ASN1_subjectPublicKeyInfo value as defined in [RFC7250], Section 3.

The OpenPGP certificate type [RFC6091] MUST NOT be used with TLS 1.3.

The server's certificate_list MUST always be non-empty. A client will send an empty certificate_list if it does not have an appropriate certificate to send in response to the server's authentication request.

4.4.2.1. OCSP Status and SCT Extensions

[RFC6066] and [RFC6961] provide extensions to negotiate the server sending OCSP responses to the client. In TLS 1.2 and below, the server replies with an empty extension to indicate negotiation of this extension and the OCSP information is carried in a CertificateStatus message. In TLS 1.3, the server's OCSP information is carried in an extension in the CertificateEntry containing the associated certificate. Specifically: The body of the "status_request" extension from the server MUST be a CertificateStatus structure as defined in [RFC6066], which is interpreted as defined in [RFC6960].

Note: status_request_v2 extension ([RFC6961]) is deprecated. TLS 1.3 servers MUST NOT act upon its presence or information in it when processing Client Hello, in particular they MUST NOT send the status_request_v2 extension in the Encrypted Extensions, Certificate Request or the Certificate messages. TLS 1.3 servers MUST be able to process Client Hello messages that include it, as it MAY be sent by clients that wish to use it in earlier protocol versions.

A server MAY request that a client present an OCSP response with its certificate by sending an empty "status_request" extension in its CertificateRequest message. If the client opts to send an OCSP response, the body of its "status_request" extension MUST be a CertificateStatus structure as defined in [RFC6066].

Similarly, [RFC6962] provides a mechanism for a server to send a Signed Certificate Timestamp (SCT) as an extension in the ServerHello in TLS 1.2 and below. In TLS 1.3, the server's SCT information is carried in an extension in CertificateEntry.

4.4.2.2. Server Certificate Selection

The following rules apply to the certificates sent by the server:

- The certificate type MUST be X.509v3 [RFC5280], unless explicitly negotiated otherwise (e.g., [RFC7250]).

- The server's end-entity certificate's public key (and associated restrictions) MUST be compatible with the selected authentication algorithm from the client's "signature_algorithms" extension (currently RSA, ECDSA, or EdDSA).

- The certificate MUST allow the key to be used for signing (i.e., the digitalSignature bit MUST be set if the Key Usage extension is present) with a signature scheme indicated in the client's "signature_algorithms"/"signature_algorithms_cert" extensions (see Section 4.2.3).

- The "server_name" [RFC6066] and "certificate_authorities" extensions are used to guide certificate selection. As servers MAY require the presence of the "server_name" extension, clients SHOULD send this extension, when applicable.

All certificates provided by the server MUST be signed by a signature algorithm advertised by the client, if it is able to provide such a chain (see Section 4.2.3). Certificates that are self-signed or certificates that are expected to be trust anchors are not validated as part of the chain and therefore MAY be signed with any algorithm.

If the server cannot produce a certificate chain that is signed only via the indicated supported algorithms, then it SHOULD continue the handshake by sending the client a certificate chain of its choice that may include algorithms that are not known to be supported by the client. This fallback chain SHOULD NOT use the deprecated SHA-1 hash algorithm in general, but MAY do so if the client's advertisement permits it, and MUST NOT do so otherwise.

If the client cannot construct an acceptable chain using the provided certificates and decides to abort the handshake, then it MUST abort the handshake with an appropriate certificate-related alert (by default, "unsupported_certificate"; see Section 6.2 for more).

If the server has multiple certificates, it chooses one of them based on the above-mentioned criteria (in addition to other criteria, such as transport layer endpoint, local configuration and preferences).

4.4.2.3. Client Certificate Selection

The following rules apply to certificates sent by the client:

- The certificate type MUST be X.509v3 [RFC5280], unless explicitly negotiated otherwise (e.g., [RFC7250]).

- If the "certificate_authorities" extension in the CertificateRequest message was present, at least one of the certificates in the certificate chain SHOULD be issued by one of the listed CAs.

- The certificates MUST be signed using an acceptable signature algorithm, as described in Section 4.3.2. Note that this relaxes the constraints on certificate-signing algorithms found in prior versions of TLS.

- If the CertificateRequest message contained a non-empty "oid_filters" extension, the end-entity certificate MUST match the extension OIDs that are recognized by the client, as described in Section 4.2.5.

Note that, as with the server certificate, there are certificates that use algorithm combinations that cannot be currently used with TLS.

4.4.2.4. Receiving a Certificate Message

In general, detailed certificate validation procedures are out of scope for TLS (see [RFC5280]). This section provides TLS-specific requirements.

If the server supplies an empty Certificate message, the client MUST abort the handshake with a "decode_error" alert.

If the client does not send any certificates (i.e., it sends an empty Certificate message), the server MAY at its discretion either continue the handshake without client authentication, or abort the handshake with a "certificate_required" alert. Also, if some aspect of the certificate chain was unacceptable (e.g., it was not signed by a known, trusted CA), the server MAY at its discretion either continue the handshake (considering the client unauthenticated) or abort the handshake.

Any endpoint receiving any certificate which it would need to validate using any signature algorithm using an MD5 hash MUST abort the handshake with a "bad_certificate" alert. SHA-1 is deprecated and it is RECOMMENDED that any endpoint receiving any certificate which it would need to validate using any signature algorithm using a SHA-1 hash abort the handshake with a "bad_certificate" alert. For clarity, this means that endpoints MAY accept these algorithms for certificates that are self-signed or are trust anchors.

All endpoints are RECOMMENDED to transition to SHA-256 or better as soon as possible to maintain interoperability with implementations currently in the process of phasing out SHA-1 support.

Note that a certificate containing a key for one signature algorithm MAY be signed using a different signature algorithm (for instance, an RSA key signed with an ECDSA key).

4.4.3. Certificate Verify

This message is used to provide explicit proof that an endpoint possesses the private key corresponding to its certificate. The CertificateVerify message also provides integrity for the handshake up to this point. Servers MUST send this message when authenticating via a certificate. Clients MUST send this message whenever authenticating via a certificate (i.e., when the Certificate message is non-empty). When sent, this message MUST appear immediately after the Certificate message and immediately prior to the Finished message.

Structure of this message:

```
struct {
    SignatureScheme algorithm;
    opaque signature<0..2^16-1>;
} CertificateVerify;
```

The algorithm field specifies the signature algorithm used (see Section 4.2.3 for the definition of this field). The signature is a digital signature using that algorithm. The content that is covered under the signature is the hash output as described in Section 4.4.1, namely:

    Transcript-Hash(Handshake Context, Certificate)

The digital signature is then computed over the concatenation of:

- A string that consists of octet 32 (0x20) repeated 64 times

- The context string

- A single 0 byte which serves as the separator

- The content to be signed

This structure is intended to prevent an attack on previous versions of TLS in which the ServerKeyExchange format meant that attackers could obtain a signature of a message with a chosen 32-byte prefix (ClientHello.random). The initial 64-byte pad clears that prefix along with the server-controlled ServerHello.random.

The context string for a server signature is: "TLS 1.3, server CertificateVerify" The context string for a client signature is: "TLS 1.3, client CertificateVerify" It is used to provide separation between signatures made in different contexts, helping against potential cross-protocol attacks.

For example, if the transcript hash was 32 bytes of 01 (this length would make sense for SHA-256), the content covered by the digital signature for a server CertificateVerify would be:

```
2020202020202020202020202020202020202020202020202020202020202020
2020202020202020202020202020202020202020202020202020202020202020
544c5320312e332c207365727665722043657274696669636174655665726966
79
00
0101010101010101010101010101010101010101010101010101010101010101
```

On the sender side the process for computing the signature field of the CertificateVerify message takes as input:

- The content covered by the digital signature

- The private signing key corresponding to the certificate sent in the previous message

If the CertificateVerify message is sent by a server, the signature algorithm MUST be one offered in the client’s "signature_algorithms" extension unless no valid certificate chain can be produced without unsupported algorithms (see Section 4.2.3).

If sent by a client, the signature algorithm used in the signature MUST be one of those present in the supported_signature_algorithms field of the "signature_algorithms" extension in the CertificateRequest message.

In addition, the signature algorithm MUST be compatible with the key in the sender's end-entity certificate. RSA signatures MUST use an RSASSA-PSS algorithm, regardless of whether RSASSA-PKCS1-v1_5 algorithms appear in "signature_algorithms". The SHA-1 algorithm MUST NOT be used in any signatures of CertificateVerify messages. All SHA-1 signature algorithms in this specification are defined solely for use in legacy certificates and are not valid for CertificateVerify signatures.

The receiver of a CertificateVerify message MUST verify the signature field. The verification process takes as input:

- The content covered by the digital signature

- The public key contained in the end-entity certificate found in the associated Certificate message.

- The digital signature received in the signature field of the CertificateVerify message

If the verification fails, the receiver MUST terminate the handshake with a "decrypt_error" alert.

4.4.4. Finished

The Finished message is the final message in the authentication block. It is essential for providing authentication of the handshake and of the computed keys.

Recipients of Finished messages MUST verify that the contents are correct and if incorrect MUST terminate the connection with a "decrypt_error" alert.

Once a side has sent its Finished message and received and validated the Finished message from its peer, it may begin to send and receive application data over the connection. There are two settings in which it is permitted to send data prior to receiving the peer's Finished:

1. Clients sending 0-RTT data as described in Section 4.2.10.

2. Servers MAY send data after sending their first flight, but because the handshake is not yet complete, they have no assurance of either the peer's identity or of its liveness (i.e., the ClientHello might have been replayed).

The key used to compute the Finished message is computed from the Base key defined in Section 4.4 using HKDF (see Section 7.1). Specifically:

```
finished_key =
    HKDF-Expand-Label(BaseKey, "finished", "", Hash.length)
```

Structure of this message:

```
struct {
    opaque verify_data[Hash.length];
} Finished;
```

The verify_data value is computed as follows:

```
verify_data =
    HMAC(finished_key,
         Transcript-Hash(Handshake Context,
                         Certificate*, CertificateVerify*))

* Only included if present.
```

HMAC [RFC2104] uses the Hash algorithm for the handshake. As noted above, the HMAC input can generally be implemented by a running hash, i.e., just the handshake hash at this point.

In previous versions of TLS, the verify_data was always 12 octets long. In TLS 1.3, it is the size of the HMAC output for the Hash used for the handshake.

Note: Alerts and any other record types are not handshake messages and are not included in the hash computations.

Any records following a Finished message MUST be encrypted under the appropriate application traffic key as described in Section 7.2. In particular, this includes any alerts sent by the server in response to client Certificate and CertificateVerify messages.

4.5. End of Early Data

```
struct {} EndOfEarlyData;
```

If the server sent an "early_data" extension, the client MUST send an EndOfEarlyData message after receiving the server Finished. If the server does not send an "early_data" extension, then the client MUST NOT send an EndOfEarlyData message. This message indicates that all 0-RTT application_data messages, if any, have been transmitted and that the following records are protected under handshake traffic keys. Servers MUST NOT send this message and clients receiving it MUST terminate the connection with an "unexpected_message" alert. This message is encrypted under keys derived from the client_early_traffic_secret.

4.6. Post-Handshake Messages

TLS also allows other messages to be sent after the main handshake. These messages use a handshake content type and are encrypted under the appropriate application traffic key.

4.6.1. New Session Ticket Message

At any time after the server has received the client Finished message, it MAY send a NewSessionTicket message. This message creates a unique association between the ticket value and a secret PSK derived from the resumption master secret (see Section 7.

The client MAY use this PSK for future handshakes by including the ticket value in the "pre_shared_key" extension in its ClientHello (Section 4.2.11). Servers MAY send multiple tickets on a single connection, either immediately after each other or after specific events (see Appendix C.4). For instance, the server might send a new ticket after post-handshake authentication in order to encapsulate the additional client authentication state. Multiple tickets are useful for clients for a variety of purposes, including:

- Opening multiple parallel HTTP connections.

- Performing connection racing across interfaces and address families via, e.g., Happy Eyeballs [RFC8305] or related techniques.

Any ticket MUST only be resumed with a cipher suite that has the same KDF hash algorithm as that used to establish the original connection.

Clients MUST only resume if the new SNI value is valid for the server certificate presented in the original session, and SHOULD only resume if the SNI value matches the one used in the original session. The latter is a performance optimization: normally, there is no reason to expect that different servers covered by a single certificate would be able to accept each other’s tickets, hence attempting resumption in that case would waste a single-use ticket. If such an indication is provided (externally or by any other means), clients MAY resume with a different SNI value.

On resumption, if reporting an SNI value to the calling application, implementations MUST use the value sent in the resumption ClientHello rather than the value sent in the previous session. Note that if a server implementation declines all PSK identities with different SNI values, these two values are always the same.

Note: Although the resumption master secret depends on the client's second flight, servers which do not request client authentication MAY compute the remainder of the transcript independently and then send a NewSessionTicket immediately upon sending its Finished rather than waiting for the client Finished. This might be appropriate in cases where the client is expected to open multiple TLS connections in parallel and would benefit from the reduced overhead of a resumption handshake, for example.

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

ticket_lifetime Indicates the lifetime in seconds as a 32-bit unsigned integer in network byte order from the time of ticket issuance. Servers MUST NOT use any value greater than 604800 seconds (7 days). The value of zero indicates that the ticket should be discarded immediately. Clients MUST NOT cache tickets for longer than 7 days, regardless of the ticket_lifetime, and MAY delete tickets earlier based on local policy. A server MAY treat a ticket as valid for a shorter period of time than what is stated in the ticket_lifetime.

ticket_age_add A securely generated, random 32-bit value that is used to obscure the age of the ticket that the client includes in the "pre_shared_key" extension. The client-side ticket age is added to this value modulo 2^32 to obtain the value that is transmitted by the client. The server MUST generate a fresh value for each ticket it sends.

ticket_nonce A per-ticket value that is unique across all tickets issued on this connection.

ticket The value of the ticket to be used as the PSK identity. The ticket itself is an opaque label. It MAY either be a database lookup key or a self-encrypted and self-authenticated value. Section 4 of [RFC5077] describes a recommended ticket construction mechanism.

extensions A set of extension values for the ticket. The "Extension" format is defined in Section 4.2. Clients MUST ignore unrecognized extensions.

The sole extension currently defined for NewSessionTicket is "early_data", indicating that the ticket may be used to send 0-RTT data (Section 4.2.10)). It contains the following value:

max_early_data_size The maximum amount of 0-RTT data that the client is allowed to send when using this ticket, in bytes. Only Application Data payload (i.e., plaintext but not padding or the inner content type byte) is counted. A server receiving more than max_early_data_size bytes of 0-RTT data SHOULD terminate the connection with an "unexpected_message" alert. Note that servers that reject early data due to lack of cryptographic material will be unable to differentiate padding from content, so clients SHOULD NOT depend on being able to send large quantities of padding in early data records.

The PSK associated with the ticket is computed as:

```
HKDF-Expand-Label(resumption_master_secret,
                  "resumption", ticket_nonce, Hash.length)
```

Because the ticket_nonce value is distinct for each NewSessionTicket message, a different PSK will be derived for each ticket.

Note that in principle it is possible to continue issuing new tickets which indefinitely extend the lifetime of the keying material originally derived from an initial non-PSK handshake (which was most likely tied to the peer’s certificate). It is RECOMMENDED that implementations place limits on the total lifetime of such keying material; these limits should take into account the lifetime of the peer’s certificate, the likelihood of intervening revocation, and the time since the peer’s online CertificateVerify signature.

4.6.2. Post-Handshake Authentication

When the client has sent the "post_handshake_auth" extension (see Section 4.2.6), a server MAY request client authentication at any time after the handshake has completed by sending a CertificateRequest message. The client MUST respond with the appropriate Authentication messages (see Section 4.4). If the client chooses to authenticate, it MUST send Certificate, CertificateVerify, and Finished. If it declines, it MUST send a Certificate message containing no certificates followed by Finished. All of the client's messages for a given response MUST appear consecutively on the wire with no intervening messages of other types.

A client that receives a CertificateRequest message without having sent the "post_handshake_auth" extension MUST send an "unexpected_message" fatal alert.

Note: Because client authentication could involve prompting the user, servers MUST be prepared for some delay, including receiving an arbitrary number of other messages between sending the CertificateRequest and receiving a response. In addition, clients which receive multiple CertificateRequests in close succession MAY respond to them in a different order than they were received (the certificate_request_context value allows the server to disambiguate the responses).

4.6.3. Key and IV Update

```
enum {
    update_not_requested(0), update_requested(1), (255)
} KeyUpdateRequest;

struct {
    KeyUpdateRequest request_update;
} KeyUpdate;
```

request_update Indicates whether the recipient of the KeyUpdate should respond with its own KeyUpdate. If an implementation receives any other value, it MUST terminate the connection with an "illegal_parameter" alert.

The KeyUpdate handshake message is used to indicate that the sender is updating its sending cryptographic keys. This message can be sent by either peer after it has sent a Finished message. Implementations that receive a KeyUpdate message prior to receiving a Finished message MUST terminate the connection with an "unexpected_message" alert. After sending a KeyUpdate message, the sender SHALL send all its traffic using the next generation of keys, computed as described in Section 7.2. Upon receiving a KeyUpdate, the receiver MUST update its receiving keys.

If the request_update field is set to "update_requested" then the receiver MUST send a KeyUpdate of its own with request_update set to "update_not_requested" prior to sending its next application data record. This mechanism allows either side to force an update to the entire connection, but causes an implementation which receives multiple KeyUpdates while it is silent to respond with a single update. Note that implementations may receive an arbitrary number of messages between sending a KeyUpdate with request_update set to update_requested and receiving the peer's KeyUpdate, because those messages may already be in flight. However, because send and receive keys are derived from independent traffic secrets, retaining the receive traffic secret does not threaten the forward secrecy of data sent before the sender changed keys.

If implementations independently send their own KeyUpdates with request_update set to "update_requested", and they cross in flight, then each side will also send a response, with the result that each side increments by two generations.

Both sender and receiver MUST encrypt their KeyUpdate messages with the old keys. Additionally, both sides MUST enforce that a KeyUpdate with the old key is received before accepting any messages encrypted with the new key. Failure to do so may allow message truncation attacks.

5. Record Protocol

The TLS record protocol takes messages to be transmitted, fragments the data into manageable blocks, protects the records, and transmits the result. Received data is verified, decrypted, reassembled, and then delivered to higher-level clients.

TLS records are typed, which allows multiple higher-level protocols to be multiplexed over the same record layer. This document specifies four content types: handshake, application data, alert, and change_cipher_spec. The change_cipher_spec record is used only for compatibility purposes (see Appendix D.4).

An implementation may receive an unencrypted record of type change_cipher_spec consisting of the single byte value 0x01 at any time after the first ClientHello message has been sent or received and before the peer's Finished message has been received and MUST simply drop it without further processing. Note that this record may appear at a point at the handshake where the implementation is expecting protected records and so it is necessary to detect this condition prior to attempting to deprotect the record. An implementation which receives any other change_cipher_spec value or which receives a protected change_cipher_spec record MUST abort the handshake with an "unexpected_message" alert. A change_cipher_spec record received before the first ClientHello message or after the peer's Finished message MUST be treated as an unexpected record type (though stateless servers may not be able to distinguish these cases from allowed cases).

Implementations MUST NOT send record types not defined in this document unless negotiated by some extension. If a TLS implementation receives an unexpected record type, it MUST terminate the connection with an "unexpected_message" alert. New record content type values are assigned by IANA in the TLS Content Type Registry as described in Section 11.

5.1. Record Layer

The record layer fragments information blocks into TLSPlaintext records carrying data in chunks of 2^14 bytes or less. Message boundaries are handled differently depending on the underlying ContentType. Any future content types MUST specify appropriate rules. Note that these rules are stricter than what was enforced in TLS 1.2.

Handshake messages MAY be coalesced into a single TLSPlaintext record or fragmented across several records, provided that:

- Handshake messages MUST NOT be interleaved with other record types. That is, if a handshake message is split over two or more records, there MUST NOT be any other records between them.

- Handshake messages MUST NOT span key changes. Implementations MUST verify that all messages immediately preceding a key change align with a record boundary; if not, then they MUST terminate the connection with an "unexpected_message" alert. Because the ClientHello, EndOfEarlyData, ServerHello, Finished, and KeyUpdate messages can immediately precede a key change, implementations MUST send these messages in alignment with a record boundary.

Implementations MUST NOT send zero-length fragments of Handshake types, even if those fragments contain padding.

Alert messages (Section 6) MUST NOT be fragmented across records and multiple Alert messages MUST NOT be coalesced into a single TLSPlaintext record. In other words, a record with an Alert type MUST contain exactly one message.

Application Data messages contain data that is opaque to TLS. Application Data messages are always protected. Zero-length fragments of Application Data MAY be sent as they are potentially useful as a traffic analysis countermeasure. Application Data fragments MAY be split across multiple records or coalesced into a single record.

```
enum {
    invalid(0),
    change_cipher_spec(20),
    alert(21),
    handshake(22),
    application_data(23),
    (255)
} ContentType;

struct {
    ContentType type;
    ProtocolVersion legacy_record_version;
    uint16 length;
    opaque fragment[TLSPlaintext.length];
} TLSPlaintext;
```

type The higher-level protocol used to process the enclosed fragment.

legacy_record_version This value MUST be set to 0x0303 for all records generated by a TLS 1.3 implementation other than an initial ClientHello (i.e., one not generated after a HelloRetryRequest), where it MAY also be 0x0301 for compatibility purposes. This field is deprecated and MUST be ignored for all purposes. Previous versions of TLS would use other values in this field under some circumstances.

length The length (in bytes) of the following TLSPlaintext.fragment. The length MUST NOT exceed 2^14 bytes. An endpoint that receives a record that exceeds this length MUST terminate the connection with a "record_overflow" alert.

fragment The data being transmitted. This value is transparent and is treated as an independent block to be dealt with by the higher-level protocol specified by the type field.

This document describes TLS 1.3, which uses the version 0x0304. This version value is historical, deriving from the use of 0x0301 for TLS 1.0 and 0x0300 for SSL 3.0. In order to maximize backwards compatibility, records containing an initial ClientHello SHOULD have version 0x0301 and a record containing a second ClientHello or a ServerHello MUST have version 0x0303, reflecting TLS 1.0 and TLS 1.2 respectively. When negotiating prior versions of TLS, endpoints follow the procedure and requirements in Appendix D.

When record protection has not yet been engaged, TLSPlaintext structures are written directly onto the wire. Once record protection has started, TLSPlaintext records are protected and sent as described in the following section. Note that application data records MUST NOT be written to the wire unprotected (see Section 2 for details).

5.2. Record Payload Protection

The record protection functions translate a TLSPlaintext structure into a TLSCiphertext. The deprotection functions reverse the process. In TLS 1.3, as opposed to previous versions of TLS, all ciphers are modeled as "Authenticated Encryption with Additional Data" (AEAD) [RFC5116]. AEAD functions provide an unified encryption and authentication operation which turns plaintext into authenticated ciphertext and back again. Each encrypted record consists of a plaintext header followed by an encrypted body, which itself contains a type and optional padding.

```
struct {
    opaque content[TLSPlaintext.length];
    ContentType type;
    uint8 zeros[length_of_padding];
} TLSInnerPlaintext;

struct {
    ContentType opaque_type = application_data; /* 23 */
    ProtocolVersion legacy_record_version = 0x0303; /* TLS v1.2 */
    uint16 length;
    opaque encrypted_record[TLSCiphertext.length];
} TLSCiphertext;
```

content The TLSPLaintext.fragment value, containing the byte encoding of a handshake or an alert message, or the raw bytes of the application’s data to send.

type The TLSPlaintext.type value containing the content type of the record.

zeros An arbitrary-length run of zero-valued bytes may appear in the cleartext after the type field. This provides an opportunity for senders to pad any TLS record by a chosen amount as long as the total stays within record size limits. See Section 5.4 for more details.

opaque_type The outer opaque_type field of a TLSCiphertext record is always set to the value 23 (application_data) for outward compatibility with middleboxes accustomed to parsing previous versions of TLS. The actual content type of the record is found in TLSInnerPlaintext.type after decryption.

legacy_record_version The legacy_record_version field is always 0x0303. TLS 1.3 TLSCiphertexts are not generated until after TLS 1.3 has been negotiated, so there are no historical compatibility concerns where other values might be received. Note that the handshake protocol including the ClientHello and ServerHello messages authenticates the protocol version, so this value is redundant.

length The length (in bytes) of the following TLSCiphertext.encrypted_record, which is the sum of the lengths of the content and the padding, plus one for the inner content type, plus any expansion added by the AEAD algorithm. The length MUST NOT exceed 2^14 + 256 bytes. An endpoint that receives a record that exceeds this length MUST terminate the connection with a "record_overflow" alert.

encrypted_record The AEAD-encrypted form of the serialized TLSInnerPlaintext structure.

AEAD algorithms take as input a single key, a nonce, a plaintext, and "additional data" to be included in the authentication check, as described in Section 2.1 of [RFC5116]. The key is either the client_write_key or the server_write_key, the nonce is derived from the sequence number and the client_write_iv or server_write_iv (see Section 5.3), and the additional data input is the record header. I.e.,

```
additional_data = TLSCiphertext.opaque_type ||
                  TLSCiphertext.legacy_record_version ||
                  TLSCiphertext.length
```

The plaintext input to the AEAD algorithm is the encoded TLSInnerPlaintext structure. Derivation of traffic keys is defined in Section 7.3.

The AEAD output consists of the ciphertext output from the AEAD encryption operation. The length of the plaintext is greater than the corresponding TLSPlaintext.length due to the inclusion of TLSInnerPlaintext.type and any padding supplied by the sender. The length of the AEAD output will generally be larger than the plaintext, but by an amount that varies with the AEAD algorithm. Since the ciphers might incorporate padding, the amount of overhead could vary with different lengths of plaintext. Symbolically,

```
AEADEncrypted =
    AEAD-Encrypt(write_key, nonce, additional_data, plaintext)
```

Then the encrypted_record field of TLSCiphertext is set to AEADEncrypted.

In order to decrypt and verify, the cipher takes as input the key, nonce, additional data, and the AEADEncrypted value. The output is either the plaintext or an error indicating that the decryption failed. There is no separate integrity check. That is:

```
plaintext of encrypted_record =
    AEAD-Decrypt(peer_write_key, nonce, additional_data, AEADEncrypted)
```

If the decryption fails, the receiver MUST terminate the connection with a "bad_record_mac" alert.

An AEAD algorithm used in TLS 1.3 MUST NOT produce an expansion greater than 255 octets. An endpoint that receives a record from its peer with TLSCiphertext.length larger than 2^14 + 256 octets MUST terminate the connection with a "record_overflow" alert. This limit is derived from the maximum TLSInnerPlaintext length of 2^14 octets + 1 octet for ContentType + the maximum AEAD expansion of 255 octets.

5.3. Per-Record Nonce

A 64-bit sequence number is maintained separately for reading and writing records. The appropriate sequence number is incremented by one after reading or writing each record. Each sequence number is set to zero at the beginning of a connection and whenever the key is changed; the first record transmitted under a particular traffic key MUST use sequence number 0.

Because the size of sequence numbers is 64-bit, they should not wrap. If a TLS implementation would need to wrap a sequence number, it MUST either re-key (Section 4.6.3) or terminate the connection.

Each AEAD algorithm will specify a range of possible lengths for the per-record nonce, from N_MIN bytes to N_MAX bytes of input ([RFC5116]). The length of the TLS per-record nonce (iv_length) is set to the larger of 8 bytes and N_MIN for the AEAD algorithm (see [RFC5116] Section 4). An AEAD algorithm where N_MAX is less than 8 bytes MUST NOT be used with TLS. The per-record nonce for the AEAD construction is formed as follows:

1. The 64-bit record sequence number is encoded in network byte order and padded to the left with zeros to iv_length.

2. The padded sequence number is XORed with the static client_write_iv or server_write_iv, depending on the role.

The resulting quantity (of length iv_length) is used as the per-record nonce.

Note: This is a different construction from that in TLS 1.2, which specified a partially explicit nonce.

5.4. Record Padding

All encrypted TLS records can be padded to inflate the size of the TLSCiphertext. This allows the sender to hide the size of the traffic from an observer.

When generating a TLSCiphertext record, implementations MAY choose to pad. An unpadded record is just a record with a padding length of zero. Padding is a string of zero-valued bytes appended to the ContentType field before encryption. Implementations MUST set the padding octets to all zeros before encrypting.

Application Data records may contain a zero-length TLSInnerPlaintext.content if the sender desires. This permits generation of plausibly-sized cover traffic in contexts where the presence or absence of activity may be sensitive. Implementations MUST NOT send Handshake or Alert records that have a zero-length TLSInnerPlaintext.content; if such a message is received, the receiving implementation MUST terminate the connection with an "unexpected_message" alert.

The padding sent is automatically verified by the record protection mechanism; upon successful decryption of a TLSCiphertext.encrypted_record, the receiving implementation scans the field from the end toward the beginning until it finds a non-zero octet. This non-zero octet is the content type of the message. This padding scheme was selected because it allows padding of any encrypted TLS record by an arbitrary size (from zero up to TLS record size limits) without introducing new content types. The design also enforces all-zero padding octets, which allows for quick detection of padding errors.

Implementations MUST limit their scanning to the cleartext returned from the AEAD decryption. If a receiving implementation does not find a non-zero octet in the cleartext, it MUST terminate the connection with an "unexpected_message" alert.

The presence of padding does not change the overall record size limitations - the full encoded TLSInnerPlaintext MUST NOT exceed 2^14 + 1 octets. If the maximum fragment length is reduced, as for example by the max_fragment_length extension from [RFC6066], then the reduced limit applies to the full plaintext, including the content type and padding.

Selecting a padding policy that suggests when and how much to pad is a complex topic and is beyond the scope of this specification. If the application layer protocol on top of TLS has its own padding, it may be preferable to pad application_data TLS records within the application layer. Padding for encrypted handshake and alert TLS records must still be handled at the TLS layer, though. Later documents may define padding selection algorithms or define a padding policy request mechanism through TLS extensions or some other means.

5.5. Limits on Key Usage

There are cryptographic limits on the amount of plaintext which can be safely encrypted under a given set of keys. [AEAD-LIMITS] provides an analysis of these limits under the assumption that the underlying primitive (AES or ChaCha20) has no weaknesses. Implementations SHOULD do a key update as described in Section 4.6.3 prior to reaching these limits.

For AES-GCM, up to 2^24.5 full-size records (about 24 million) may be encrypted on a given connection while keeping a safety margin of approximately 2^-57 for Authenticated Encryption (AE) security. For ChaCha20/Poly1305, the record sequence number would wrap before the safety limit is reached.

6. Alert Protocol

One of the content types supported by the TLS record layer is the alert type. Like other messages, alert messages are encrypted as specified by the current connection state.

Alert messages convey a description of the alert and a legacy field that conveyed the severity of the message in previous versions of TLS. Alerts are divided into two classes: closure alerts and error alerts. In TLS 1.3, the severity is implicit in the type of alert being sent, and the 'level' field can safely be ignored. The "close_notify" alert is used to indicate orderly closure of one direction of the connection. Upon receiving such an alert, the TLS implementation SHOULD indicate end-of-data to the application.

Error alerts indicate abortive closure of the connection (see Section 6.2). Upon receiving an error alert, the TLS implementation SHOULD indicate an error to the application and MUST NOT allow any further data to be sent or received on the connection. Servers and clients MUST forget the secret values and keys established in failed connections, with the exception of the PSKs associated with session tickets, which SHOULD be discarded if possible.

All the alerts listed in Section 6.2 MUST be sent with AlertLevel=fatal and MUST be treated as error alerts regardless of the AlertLevel in the message. Unknown alert types MUST be treated as error alerts.

Note: TLS defines two generic alerts (see Section 6) to use upon failure to parse a message. Peers which receive a message which cannot be parsed according to the syntax (e.g., have a length extending beyond the message boundary or contain an out-of-range length) MUST terminate the connection with a "decode_error" alert. Peers which receive a message which is syntactically correct but semantically invalid (e.g., a DHE share of p - 1, or an invalid enum) MUST terminate the connection with an "illegal_parameter" alert.

```
enum { warning(1), fatal(2), (255) } AlertLevel;

enum {
    close_notify(0),
    unexpected_message(10),
    bad_record_mac(20),
    record_overflow(22),
    handshake_failure(40),
    bad_certificate(42),
    unsupported_certificate(43),
    certificate_revoked(44),
    certificate_expired(45),
    certificate_unknown(46),
    illegal_parameter(47),
    unknown_ca(48),
    access_denied(49),
    decode_error(50),
    decrypt_error(51),
    protocol_version(70),
    insufficient_security(71),
    internal_error(80),
    inappropriate_fallback(86),
    user_canceled(90),
    missing_extension(109),
    unsupported_extension(110),
    unrecognized_name(112),
    bad_certificate_status_response(113),
    unknown_psk_identity(115),
    certificate_required(116),
    no_application_protocol(120),
    (255)
} AlertDescription;

struct {
    AlertLevel level;
    AlertDescription description;
} Alert;
```

6.1. Closure Alerts

The client and the server must share knowledge that the connection is ending in order to avoid a truncation attack.

close_notify This alert notifies the recipient that the sender will not send any more messages on this connection. Any data received after a closure alert has been received MUST be ignored.

user_canceled This alert notifies the recipient that the sender is canceling the handshake for some reason unrelated to a protocol failure. If a user cancels an operation after the handshake is complete, just closing the connection by sending a "close_notify" is more appropriate. This alert SHOULD be followed by a "close_notify". This alert generally has AlertLevel=warning.

Either party MAY initiate a close of its write side of the connection by sending a "close_notify" alert. Any data received after a closure alert has been received MUST be ignored. If a transport-level close is received prior to a "close_notify", the receiver cannot know that all the data that was sent has been received.

Each party MUST send a "close_notify" alert before closing its write side of the connection, unless it has already sent some error alert. This does not have any effect on its read side of the connection. Note that this is a change from versions of TLS prior to TLS 1.3 in which implementations were required to react to a "close_notify" by discarding pending writes and sending an immediate "close_notify" alert of their own. That previous requirement could cause truncation in the read side. Both parties need not wait to receive a "close_notify" alert before closing their read side of the connection, though doing so would introduce the possibility of truncation.

If the application protocol using TLS provides that any data may be carried over the underlying transport after the TLS connection is closed, the TLS implementation MUST receive a "close_notify" alert before indicating end-of-data to the application-layer. No part of this standard should be taken to dictate the manner in which a usage profile for TLS manages its data transport, including when connections are opened or closed.

Note: It is assumed that closing the write side of a connection reliably delivers pending data before destroying the transport.

6.2. Error Alerts

Error handling in the TLS Handshake Protocol is very simple. When an error is detected, the detecting party sends a message to its peer. Upon transmission or receipt of a fatal alert message, both parties MUST immediately close the connection.

Whenever an implementation encounters a fatal error condition, it SHOULD send an appropriate fatal alert and MUST close the connection without sending or receiving any additional data. In the rest of this specification, when the phrases "terminate the connection" and "abort the handshake" are used without a specific alert it means that the implementation SHOULD send the alert indicated by the descriptions below. The phrases "terminate the connection with a X alert" and "abort the handshake with a X alert" mean that the implementation MUST send alert X if it sends any alert. All alerts defined in this section below, as well as all unknown alerts, are universally considered fatal as of TLS 1.3 (see Section 6). The implementation SHOULD provide a way to facilitate logging the sending and receiving of alerts.

The following error alerts are defined:

unexpected_message An inappropriate message (e.g., the wrong handshake message, premature application data, etc.) was received. This alert should never be observed in communication between proper implementations.

bad_record_mac This alert is returned if a record is received which cannot be deprotected. Because AEAD algorithms combine decryption and verification, and also to avoid side channel attacks, this alert is used for all deprotection failures. This alert should never be observed in communication between proper implementations, except when messages were corrupted in the network.

record_overflow A TLSCiphertext record was received that had a length more than 2^14 + 256 bytes, or a record decrypted to a TLSPlaintext record with more than 2^14 bytes (or some other negotiated limit). This alert should never be observed in communication between proper implementations, except when messages were corrupted in the network.

handshake_failure Receipt of a "handshake_failure" alert message indicates that the sender was unable to negotiate an acceptable set of security parameters given the options available.

bad_certificate A certificate was corrupt, contained signatures that did not verify correctly, etc.

unsupported_certificate A certificate was of an unsupported type.

certificate_revoked A certificate was revoked by its signer.

certificate_expired A certificate has expired or is not currently valid.

certificate_unknown Some other (unspecified) issue arose in processing the certificate, rendering it unacceptable.

illegal_parameter A field in the handshake was incorrect or inconsistent with other fields. This alert is used for errors which conform to the formal protocol syntax but are otherwise incorrect.

unknown_ca A valid certificate chain or partial chain was received, but the certificate was not accepted because the CA certificate could not be located or could not be matched with a known trust anchor.

access_denied A valid certificate or PSK was received, but when access control was applied, the sender decided not to proceed with negotiation.

decode_error A message could not be decoded because some field was out of the specified range or the length of the message was incorrect. This alert is used for errors where the message does not conform to the formal protocol syntax. This alert should never be observed in communication between proper implementations, except when messages were corrupted in the network.

decrypt_error A handshake (not record-layer) cryptographic operation failed, including being unable to correctly verify a signature or validate a Finished message or a PSK binder.

protocol_version The protocol version the peer has attempted to negotiate is recognized but not supported. (see Appendix D)

insufficient_security Returned instead of "handshake_failure" when a negotiation has failed specifically because the server requires parameters more secure than those supported by the client.

internal_error An internal error unrelated to the peer or the correctness of the protocol (such as a memory allocation failure) makes it impossible to continue.

inappropriate_fallback Sent by a server in response to an invalid connection retry attempt from a client (see [RFC7507]).

missing_extension Sent by endpoints that receive a handshake message not containing an extension that is mandatory to send for the offered TLS version or other negotiated parameters.

unsupported_extension Sent by endpoints receiving any handshake message containing an extension known to be prohibited for inclusion in the given handshake message, or including any extensions in a ServerHello or Certificate not first offered in the corresponding ClientHello.

unrecognized_name Sent by servers when no server exists identified by the name provided by the client via the "server_name" extension (see [RFC6066]).

bad_certificate_status_response Sent by clients when an invalid or unacceptable OCSP response is provided by the server via the "status_request" extension (see [RFC6066]).

unknown_psk_identity Sent by servers when PSK key establishment is desired but no acceptable PSK identity is provided by the client. Sending this alert is OPTIONAL; servers MAY instead choose to send a "decrypt_error" alert to merely indicate an invalid PSK identity.

certificate_required Sent by servers when a client certificate is desired but none was provided by the client.

no_application_protocol Sent by servers when a client "application_layer_protocol_negotiation" extension advertises only protocols that the server does not support (see [RFC7301]).

New Alert values are assigned by IANA as described in Section 11.

7. Cryptographic Computations

The TLS handshake establishes one or more input secrets which are combined to create the actual working keying material, as detailed below. The key derivation process incorporates both the input secrets and the handshake transcript. Note that because the handshake transcript includes the random values from the Hello messages, any given handshake will have different traffic secrets, even if the same input secrets are used, as is the case when the same PSK is used for multiple connections.

7.1. Key Schedule

The key derivation process makes use of the HKDF-Extract and HKDF-Expand functions as defined for HKDF [RFC5869], as well as the functions defined below:

```
HKDF-Expand-Label(Secret, Label, Context, Length) =
     HKDF-Expand(Secret, HkdfLabel, Length)

Where HkdfLabel is specified as:

struct {
    uint16 length = Length;
    opaque label<7..255> = "tls13 " + Label;
    opaque context<0..255> = Context;
} HkdfLabel;

Derive-Secret(Secret, Label, Messages) =
     HKDF-Expand-Label(Secret, Label,
                       Transcript-Hash(Messages), Hash.length)
```

The Hash function used by Transcript-Hash and HKDF is the cipher suite hash algorithm. Hash.length is its output length in bytes. Messages is the concatenation of the indicated handshake messages, including the handshake message type and length fields, but not including record layer headers. Note that in some cases a zero-length Context (indicated by "") is passed to HKDF-Expand-Label. The Labels specified in this document are all ASCII strings, and do not include a trailing NUL byte.

Note: with common hash functions, any label longer than 12 characters requires an additional iteration of the hash function to compute. The labels in this specification have all been chosen to fit within this limit.

Keys are derived from two input secrets using the HKDF-Extract and Derive-Secret functions. The general pattern for adding a new secret is to use HKDF-Extract with the salt being the current secret state and the IKM being the new secret to be added. In this version of TLS 1.3, the two input secrets are:

- PSK (a pre-shared key established externally or derived from the resumption_master_secret value from a previous connection)

- (EC)DHE shared secret (Section 7.4)

This produces a full key derivation schedule shown in the diagram below. In this diagram, the following formatting conventions apply:

- HKDF-Extract is drawn as taking the Salt argument from the top and the IKM argument from the left, with its output to the bottom and the name of the output on the right.

- Derive-Secret's Secret argument is indicated by the incoming arrow. For instance, the Early Secret is the Secret for generating the client_early_traffic_secret.

- "0" indicates a string of Hash-lengths bytes set to 0.

```
                 0
                 |
                 v
   PSK ->  HKDF-Extract = Early Secret
                 |
                 +-----> Derive-Secret(.,
                 |                     "ext binder" |
                 |                     "res binder",
                 |                     "")
                 |                     = binder_key
                 |
                 +-----> Derive-Secret(., "c e traffic",
                 |                     ClientHello)
                 |                     = client_early_traffic_secret
                 |
                 +-----> Derive-Secret(., "e exp master",
                 |                     ClientHello)
                 |                     = early_exporter_master_secret
                 v
           Derive-Secret(., "derived", "")
                 |
                 v
(EC)DHE -> HKDF-Extract = Handshake Secret
                 |
                 +-----> Derive-Secret(., "c hs traffic",
                 |                     ClientHello...ServerHello)
                 |                     = client_handshake_traffic_secret
                 |
                 +-----> Derive-Secret(., "s hs traffic",
                 |                     ClientHello...ServerHello)
                 |                     = server_handshake_traffic_secret
                 v
           Derive-Secret(., "derived", "")
                 |
                 v
      0 -> HKDF-Extract = Master Secret
                 |
                 +-----> Derive-Secret(., "c ap traffic",
                 |                     ClientHello...server Finished)
                 |                     = client_application_traffic_secret_0
                 |
                 +-----> Derive-Secret(., "s ap traffic",
```

```
                 |                     ClientHello...server Finished)
                 |                     = server_application_traffic_secret_0
                 |
                 +-----> Derive-Secret(., "exp master",
                 |                     ClientHello...server Finished)
                 |                     = exporter_master_secret
                 |
                 +-----> Derive-Secret(., "res master",
                                       ClientHello...client Finished)
                                       = resumption_master_secret
```

The general pattern here is that the secrets shown down the left side of the diagram are just raw entropy without context, whereas the secrets down the right side include handshake context and therefore can be used to derive working keys without additional context. Note that the different calls to Derive-Secret may take different Messages arguments, even with the same secret. In a 0-RTT exchange, Derive-Secret is called with four distinct transcripts; in a 1-RTT-only exchange with three distinct transcripts.

If a given secret is not available, then the 0-value consisting of a string of Hash.length bytes set to zeros is used. Note that this does not mean skipping rounds, so if PSK is not in use Early Secret will still be HKDF-Extract(0, 0). For the computation of the binder_secret, the label is "ext binder" for external PSKs (those provisioned outside of TLS) and "res binder" for resumption PSKs (those provisioned as the resumption master secret of a previous handshake). The different labels prevent the substitution of one type of PSK for the other.

There are multiple potential Early Secret values depending on which PSK the server ultimately selects. The client will need to compute one for each potential PSK; if no PSK is selected, it will then need to compute the early secret corresponding to the zero PSK.

Once all the values which are to be derived from a given secret have been computed, that secret SHOULD be erased.

7.2. Updating Traffic Secrets

Once the handshake is complete, it is possible for either side to update its sending traffic keys using the KeyUpdate handshake message defined in Section 4.6.3. The next generation of traffic keys is computed by generating client_/server_application_traffic_secret_N+1 from client_/server_application_traffic_secret_N as described in this section then re-deriving the traffic keys as described in Section 7.3.

The next-generation application_traffic_secret is computed as:

```
application_traffic_secret_N+1 =
    HKDF-Expand-Label(application_traffic_secret_N,
                      "traffic upd", "", Hash.length)
```

Once client/server_application_traffic_secret_N+1 and its associated traffic keys have been computed, implementations SHOULD delete client_/server_application_traffic_secret_N and its associated traffic keys.

7.3. Traffic Key Calculation

The traffic keying material is generated from the following input values:

- A secret value

- A purpose value indicating the specific value being generated

- The length of the key being generated

The traffic keying material is generated from an input traffic secret value using:

```
[sender]_write_key = HKDF-Expand-Label(Secret, "key", "", key_length)
[sender]_write_iv  = HKDF-Expand-Label(Secret, "iv" , "", iv_length)
```

[sender] denotes the sending side. The Secret value for each record type is shown in the table below.

Record Type	Secret
0-RTT Application	client_early_traffic_secret
Handshake	[sender]_handshake_traffic_secret
Application Data	[sender]_application_traffic_secret_N

All the traffic keying material is recomputed whenever the underlying Secret changes (e.g., when changing from the handshake to application data keys or upon a key update).

7.4. (EC)DHE Shared Secret Calculation

7.4.1. Finite Field Diffie-Hellman

For finite field groups, a conventional Diffie-Hellman [DH76] computation is performed. The negotiated key (Z) is converted to a byte string by encoding in big-endian and left padded with zeros up to the size of the prime. This byte string is used as the shared secret in the key schedule as specified above.

Note that this construction differs from previous versions of TLS which remove leading zeros.

7.4.2. Elliptic Curve Diffie-Hellman

For secp256r1, secp384r1 and secp521r1, ECDH calculations (including parameter and key generation as well as the shared secret calculation) are performed according to [IEEE1363] using the ECKAS-DH1 scheme with the identity map as key derivation function (KDF), so that the shared secret is the x-coordinate of the ECDH shared secret elliptic curve point represented as an octet string. Note that this octet string (Z in IEEE 1363 terminology) as output by FE2OSP, the Field Element to Octet String Conversion Primitive, has constant length for any given field; leading zeros found in this octet string MUST NOT be truncated.

(Note that this use of the identity KDF is a technicality. The complete picture is that ECDH is employed with a non-trivial KDF because TLS does not directly use this secret for anything other than for computing other secrets.)

ECDH functions are used as follows:

- The public key to put into the KeyShareEntry.key_exchange structure is the result of applying the ECDH scalar multiplication function to the secret key of appropriate length (into scalar input) and the standard public basepoint (into u-coordinate point input).

- The ECDH shared secret is the result of applying the ECDH scalar multiplication function to the secret key (into scalar input) and the peer’s public key (into u-coordinate point input). The output is used raw, with no processing.

For X25519 and X448, implementations SHOULD use the approach specified in [RFC7748] to calculate the Diffie-Hellman shared secret. Implementations MUST check whether the computed Diffie-Hellman shared secret is the all-zero value and abort if so, as described in

Section 6 of [RFC7748]. If implementors use an alternative implementation of these elliptic curves, they SHOULD perform the additional checks specified in Section 7 of [RFC7748].

7.5. Exporters

[RFC5705] defines keying material exporters for TLS in terms of the TLS pseudorandom function (PRF). This document replaces the PRF with HKDF, thus requiring a new construction. The exporter interface remains the same.

The exporter value is computed as:

```
TLS-Exporter(label, context_value, key_length) =
    HKDF-Expand-Label(Derive-Secret(Secret, label, ""),
                      "exporter", Hash(context_value), key_length)
```

Where Secret is either the early_exporter_master_secret or the exporter_master_secret. Implementations MUST use the exporter_master_secret unless explicitly specified by the application. The early_exporter_master_secret is defined for use in settings where an exporter is needed for 0-RTT data. A separate interface for the early exporter is RECOMMENDED; this avoids the exporter user accidentally using an early exporter when a regular one is desired or vice versa.

If no context is provided, the context_value is zero-length. Consequently, providing no context computes the same value as providing an empty context. This is a change from previous versions of TLS where an empty context produced a different output to an absent context. As of this document's publication, no allocated exporter label is used both with and without a context. Future specifications MUST NOT define a use of exporters that permit both an empty context and no context with the same label. New uses of exporters SHOULD provide a context in all exporter computations, though the value could be empty.

Requirements for the format of exporter labels are defined in section 4 of [RFC5705].

8. 0-RTT and Anti-Replay

As noted in Section 2.3 and Appendix E.5, TLS does not provide inherent replay protections for 0-RTT data. There are two potential threats to be concerned with:

- Network attackers who mount a replay attack by simply duplicating a flight of 0-RTT data.

- Network attackers who take advantage of client retry behavior to arrange for the server to receive multiple copies of an application message. This threat already exists to some extent because clients that value robustness respond to network errors by attempting to retry requests. However, 0-RTT adds an additional dimension for any server system which does not maintain globally consistent server state. Specifically, if a server system has multiple zones where tickets from zone A will not be accepted in zone B, then an attacker can duplicate a ClientHello and early data intended for A to both A and B. At A, the data will be accepted in 0-RTT, but at B the server will reject 0-RTT data and instead force a full handshake. If the attacker blocks the ServerHello from A, then the client will complete the handshake with B and probably retry the request, leading to duplication on the server system as a whole.

The first class of attack can be prevented by sharing state to guarantee that the 0-RTT data is accepted at most once. Servers SHOULD provide that level of replay safety, by implementing one of the methods described in this section or by equivalent means. It is understood, however, that due to operational concerns not all deployments will maintain state at that level. Therefore, in normal operation, clients will not know which, if any, of these mechanisms servers actually implement and hence MUST only send early data which they deem safe to be replayed.

In addition to the direct effects of replays, there is a class of attacks where even operations normally considered idempotent could be exploited by a large number of replays (timing attacks, resource limit exhaustion and others described in Appendix E.5). Those can be mitigated by ensuring that every 0-RTT payload can be replayed only a limited number of times. The server MUST ensure that any instance of it (be it a machine, a thread or any other entity within the relevant serving infrastructure) would accept 0-RTT for the same 0-RTT handshake at most once; this limits the number of replays to the number of server instances in the deployment. Such a guarantee can be accomplished by locally recording data from recently-received ClientHellos and rejecting repeats, or by any other method that provides the same or a stronger guarantee. The "at most once per server instance" guarantee is a minimum requirement; servers SHOULD limit 0-RTT replays further when feasible.

The second class of attack cannot be prevented at the TLS layer and MUST be dealt with by any application. Note that any application whose clients implement any kind of retry behavior already needs to implement some sort of anti-replay defense.

8.1. Single-Use Tickets

The simplest form of anti-replay defense is for the server to only allow each session ticket to be used once. For instance, the server can maintain a database of all outstanding valid tickets; deleting each ticket from the database as it is used. If an unknown ticket is provided, the server would then fall back to a full handshake.

If the tickets are not self-contained but rather are database keys, and the corresponding PSKs are deleted upon use, then connections established using PSKs enjoy forward secrecy. This improves security for all 0-RTT data and PSK usage when PSK is used without (EC)DHE.

Because this mechanism requires sharing the session database between server nodes in environments with multiple distributed servers, it may be hard to achieve high rates of successful PSK 0-RTT connections when compared to self-encrypted tickets. Unlike session databases, session tickets can successfully do PSK-based session establishment even without consistent storage, though when 0-RTT is allowed they still require consistent storage for anti-replay of 0-RTT data, as detailed in the following section.

8.2. Client Hello Recording

An alternative form of anti-replay is to record a unique value derived from the ClientHello (generally either the random value or the PSK binder) and reject duplicates. Recording all ClientHellos causes state to grow without bound, but a server can instead record ClientHellos within a given time window and use the "obfuscated_ticket_age" to ensure that tickets aren’t reused outside that window.

In order to implement this, when a ClientHello is received, the server first verifies the PSK binder as described Section 4.2.11. It then computes the expected_arrival_time as described in the next section and rejects 0-RTT if it is outside the recording window, falling back to the 1-RTT handshake.

If the expected arrival time is in the window, then the server checks to see if it has recorded a matching ClientHello. If one is found, it either aborts the handshake with an "illegal_parameter" alert or accepts the PSK but reject 0-RTT. If no matching ClientHello is found, then it accepts 0-RTT and then stores the ClientHello for as long as the expected_arrival_time is inside the window. Servers MAY also implement data stores with false positives, such as Bloom filters, in which case they MUST respond to apparent replay by rejecting 0-RTT but MUST NOT abort the handshake.

The server MUST derive the storage key only from validated sections of the ClientHello. If the ClientHello contains multiple PSK identities, then an attacker can create multiple ClientHellos with different binder values for the less-preferred identity on the assumption that the server will not verify it, as recommended by Section 4.2.11. I.e., if the client sends PSKs A and B but the server prefers A, then the attacker can change the binder for B without affecting the binder for A. If the binder for B is part of the storage key, then this ClientHello will not appear as a duplicate, which will cause the ClientHello to be accepted, and may cause side effects such as replay cache pollution, although any 0-RTT data will not be decryptable because it will use different keys. If the validated binder or the ClientHello.random are used as the storage key, then this attack is not possible.

Because this mechanism does not require storing all outstanding tickets, it may be easier to implement in distributed systems with high rates of resumption and 0-RTT, at the cost of potentially weaker anti-replay defense because of the difficulty of reliably storing and retrieving the received ClientHello messages. In many such systems, it is impractical to have globally consistent storage of all the received ClientHellos. In this case, the best anti-replay protection is provided by having a single storage zone be authoritative for a given ticket and refusing 0-RTT for that ticket in any other zone. This approach prevents simple replay by the attacker because only one zone will accept 0-RTT data. A weaker design is to implement separate storage for each zone but allow 0-RTT in any zone. This approach limits the number of replays to once per zone. Application message duplication of course remains possible with either design.

When implementations are freshly started, they SHOULD reject 0-RTT as long as any portion of their recording window overlaps the startup time. Otherwise, they run the risk of accepting replays which were originally sent during that period.

Note: If the client’s clock is running much faster than the server’s then a ClientHello may be received that is outside the window in the future, in which case it might be accepted for 1-RTT, causing a client retry, and then acceptable later for 0-RTT. This is another variant of the second form of attack described above.

8.3. Freshness Checks

Because the ClientHello indicates the time at which the client sent it, it is possible to efficiently determine whether a ClientHello was likely sent reasonably recently and only accept 0-RTT for such a ClientHello, otherwise falling back to a 1-RTT handshake. This is necessary for the ClientHello storage mechanism described in

Section 8.2 because otherwise the server needs to store an unlimited number of ClientHellos and is a useful optimization for self-contained single-use tickets because it allows efficient rejection of ClientHellos which cannot be used for 0-RTT.

In order to implement this mechanism, a server needs to store the time that the server generated the session ticket, offset by an estimate of the round trip time between client and server. I.e.,

```
    adjusted_creation_time = creation_time + estimated_RTT
```

This value can be encoded in the ticket, thus avoiding the need to keep state for each outstanding ticket. The server can determine the client’s view of the age of the ticket by subtracting the ticket’s "ticket_age_add value" from the "obfuscated_ticket_age" parameter in the client’s "pre_shared_key" extension. The server can determine the "expected arrival time" of the ClientHello as:

```
  expected_arrival_time = adjusted_creation_time + clients_ticket_age
```

When a new ClientHello is received, the expected_arrival_time is then compared against the current server wall clock time and if they differ by more than a certain amount, 0-RTT is rejected, though the 1-RTT handshake can be allowed to complete.

There are several potential sources of error that might cause mismatches between the expected arrival time and the measured time. Variations in client and server clock rates are likely to be minimal, though potentially the absolute times may be off by large values. Network propagation delays are the most likely causes of a mismatch in legitimate values for elapsed time. Both the NewSessionTicket and ClientHello messages might be retransmitted and therefore delayed, which might be hidden by TCP. For clients on the Internet, this implies windows on the order of ten seconds to account for errors in clocks and variations in measurements; other deployment scenarios may have different needs. Clock skew distributions are not symmetric, so the optimal tradeoff may involve an asymmetric range of permissible mismatch values.

Note that freshness checking alone is not sufficient to prevent replays because it does not detect them during the error window, which, depending on bandwidth and system capacity could include billions of replays in real-world settings. In addition, this freshness checking is only done at the time the ClientHello is received, and not when later early application data records are received. After early data is accepted, records may continue to be streamed to the server over a longer time period.

9. Compliance Requirements

9.1. Mandatory-to-Implement Cipher Suites

In the absence of an application profile standard specifying otherwise, a TLS-compliant application MUST implement the TLS_AES_128_GCM_SHA256 [GCM] cipher suite and SHOULD implement the TLS_AES_256_GCM_SHA384 [GCM] and TLS_CHACHA20_POLY1305_SHA256 [RFC7539] cipher suites. (see Appendix B.4)

A TLS-compliant application MUST support digital signatures with rsa_pkcs1_sha256 (for certificates), rsa_pss_rsae_sha256 (for CertificateVerify and certificates), and ecdsa_secp256r1_sha256. A TLS-compliant application MUST support key exchange with secp256r1 (NIST P-256) and SHOULD support key exchange with X25519 [RFC7748].

9.2. Mandatory-to-Implement Extensions

In the absence of an application profile standard specifying otherwise, a TLS-compliant application MUST implement the following TLS extensions:

- Supported Versions ("supported_versions"; Section 4.2.1)

- Cookie ("cookie"; Section 4.2.2)

- Signature Algorithms ("signature_algorithms"; Section 4.2.3)

- Signature Algorithms Certificate ("signature_algorithms_cert"; Section 4.2.3)

- Negotiated Groups ("supported_groups"; Section 4.2.7)

- Key Share ("key_share"; Section 4.2.8)

- Server Name Indication ("server_name"; Section 3 of [RFC6066])

All implementations MUST send and use these extensions when offering applicable features:

- "supported_versions" is REQUIRED for all ClientHello, ServerHello and HelloRetryRequest messages.

- "signature_algorithms" is REQUIRED for certificate authentication.

- "supported_groups" is REQUIRED for ClientHello messages using DHE or ECDHE key exchange.

- "key_share" is REQUIRED for DHE or ECDHE key exchange.

- "pre_shared_key" is REQUIRED for PSK key agreement.

- "psk_key_exchange_modes" is REQUIRED for PSK key agreement.

A client is considered to be attempting to negotiate using this specification if the ClientHello contains a "supported_versions" extension with 0x0304 contained in its body. Such a ClientHello message MUST meet the following requirements:

- If not containing a "pre_shared_key" extension, it MUST contain both a "signature_algorithms" extension and a "supported_groups" extension.

- If containing a "supported_groups" extension, it MUST also contain a "key_share" extension, and vice versa. An empty KeyShare.client_shares vector is permitted.

Servers receiving a ClientHello which does not conform to these requirements MUST abort the handshake with a "missing_extension" alert.

Additionally, all implementations MUST support use of the "server_name" extension with applications capable of using it. Servers MAY require clients to send a valid "server_name" extension. Servers requiring this extension SHOULD respond to a ClientHello lacking a "server_name" extension by terminating the connection with a "missing_extension" alert.

9.3. Protocol Invariants

This section describes invariants that TLS endpoints and middleboxes MUST follow. It also applies to earlier versions of TLS.

TLS is designed to be securely and compatibly extensible. Newer clients or servers, when communicating with newer peers, should negotiate the most preferred common parameters. The TLS handshake provides downgrade protection: Middleboxes passing traffic between a newer client and newer server without terminating TLS should be unable to influence the handshake (see Appendix E.1). At the same time, deployments update at different rates, so a newer client or server MAY continue to support older parameters, which would allow it to interoperate with older endpoints.

For this to work, implementations MUST correctly handle extensible fields:

- A client sending a ClientHello MUST support all parameters advertised in it. Otherwise, the server may fail to interoperate by selecting one of those parameters.

- A server receiving a ClientHello MUST correctly ignore all unrecognized cipher suites, extensions, and other parameters. Otherwise, it may fail to interoperate with newer clients. In TLS 1.3, a client receiving a CertificateRequest or NewSessionTicket MUST also ignore all unrecognized extensions.

- A middlebox which terminates a TLS connection MUST behave as a compliant TLS server (to the original client), including having a certificate which the client is willing to accept, and as a compliant TLS client (to the original server), including verifying the original server's certificate. In particular, it MUST generate its own ClientHello containing only parameters it understands, and it MUST generate a fresh ServerHello random value, rather than forwarding the endpoint's value.

  Note that TLS's protocol requirements and security analysis only apply to the two connections separately. Safely deploying a TLS terminator requires additional security considerations which are beyond the scope of this document.

- An middlebox which forwards ClientHello parameters it does not understand MUST NOT process any messages beyond that ClientHello. It MUST forward all subsequent traffic unmodified. Otherwise, it may fail to interoperate with newer clients and servers.

  Forwarded ClientHellos may contain advertisements for features not supported by the middlebox, so the response may include future TLS additions the middlebox does not recognize. These additions MAY change any message beyond the ClientHello arbitrarily. In particular, the values sent in the ServerHello might change, the ServerHello format might change, and the TLSCiphertext format might change.

The design of TLS 1.3 was constrained by widely-deployed non-compliant TLS middleboxes (see Appendix D.4), however it does not relax the invariants. Those middleboxes continue to be non-compliant.

10. Security Considerations

Security issues are discussed throughout this memo, especially in Appendix C, Appendix D, and Appendix E.

11. IANA Considerations

This document uses several registries that were originally created in [RFC4346]. IANA [SHALL update/has updated] these to reference this document. The registries and their allocation policies are below:

- TLS Cipher Suite Registry: values with the first byte in the range 0-254 (decimal) are assigned via Specification Required [RFC8126]. Values with the first byte 255 (decimal) are reserved for Private Use [RFC8126].

  IANA [SHALL add/has added] the cipher suites listed in Appendix B.4 to the registry. The "Value" and "Description" columns are taken from the table. The "DTLS-OK" and "Recommended" columns are both marked as "Yes" for each new cipher suite. [[This assumes [I-D.ietf-tls-iana-registry-updates] has been applied.]]

- TLS ContentType Registry: Future values are allocated via Standards Action [RFC8126].

- TLS Alert Registry: Future values are allocated via Standards Action [RFC8126]. IANA [SHALL update/has updated] this registry to include values for "missing_extension" and "certificate_required". The "DTLS-OK" column is marked as "Yes" for each new alert.

- TLS HandshakeType Registry: Future values are allocated via Standards Action [RFC8126]. IANA [SHALL update/has updated] this registry to rename item 4 from "NewSessionTicket" to "new_session_ticket" and to add the "hello_retry_request_RESERVED", "encrypted_extensions", "end_of_early_data", "key_update", and "message_hash" values. The "DTLS-OK" are marked as "Yes" for each of these additions.

This document also uses the TLS ExtensionType Registry originally created in [RFC4366]. IANA has updated it to reference this document. Changes to the registry follow:

- IANA [SHALL update/has updated] the registration policy as follows:

  Values with the first byte in the range 0-254 (decimal) are assigned via Specification Required [RFC8126]. Values with the first byte 255 (decimal) are reserved for Private Use [RFC8126].

- IANA [SHALL update/has updated] this registry to include the "key_share", "pre_shared_key", "psk_key_exchange_modes", "early_data", "cookie", "supported_versions", "certificate_authorities", "oid_filters", "post_handshake_auth", and "signature_algorithms_cert", extensions with the values defined in this document and the Recommended value of "Yes".

- IANA [SHALL update/has updated] this registry to include a "TLS 1.3" column which lists the messages in which the extension may appear. This column [SHALL be/has been] initially populated from the table in Section 4.2 with any extension not listed there marked as "-" to indicate that it is not used by TLS 1.3.

In addition, this document defines two new registries to be maintained by IANA:

- TLS SignatureScheme Registry: Values with the first byte in the range 0-253 (decimal) are assigned via Specification Required [RFC8126]. Values with the first byte 254 or 255 (decimal) are reserved for Private Use [RFC8126]. Values with the first byte in the range 0-6 or with the second byte in the range 0-3 that are not currently allocated are reserved for backwards compatibility. This registry SHALL have a "Recommended" column. The registry [shall be/ has been] initially populated with the values described in Section 4.2.3. The following values SHALL be marked as "Recommended": ecdsa_secp256r1_sha256, ecdsa_secp384r1_sha384, rsa_pss_rsae_sha256, rsa_pss_rsae_sha384, rsa_pss_rsae_sha512, rsa_pss_pss_sha256, rsa_pss_pss_sha384, rsa_pss_pss_sha512, and ed25519.

- TLS PskKeyExchangeMode Registry: Values in the range 0-253 (decimal) are assigned via Specification Required [RFC8126]. Values with the first byte 254 or 255 (decimal) are reserved for Private Use [RFC8126]. This registry SHALL have a "Recommended" column. The registry [shall be/ has been] initially populated psk_ke (0) and psk_dhe_ke (1). Both SHALL be marked as "Recommended".

12. References

12.1. Normative References

[DH] Diffie, W. and M. Hellman, "New Directions in Cryptography", IEEE Transactions on Information Theory, V.IT-22 n.6 , June 1977.

[DH76] Diffie, W. and M. Hellman, "New directions in cryptography", IEEE Transactions on Information Theory Vol. 22, pp. 644-654, DOI 10.1109/tit.1976.1055638, November 1976.

[GCM] Dworkin, M., "Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC", NIST Special Publication 800-38D, November 2007.

[RFC2104] Krawczyk, H., Bellare, M., and R. Canetti, "HMAC: Keyed-Hashing for Message Authentication", RFC 2104, DOI 10.17487/RFC2104, February 1997, <https://www.rfc-editor.org/info/rfc2104>.

[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>.

[RFC5116] McGrew, D., "An Interface and Algorithms for Authenticated Encryption", RFC 5116, DOI 10.17487/RFC5116, January 2008, <https://www.rfc-editor.org/info/rfc5116>.

[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, <https://www.rfc-editor.org/info/rfc5280>.

[RFC5705] Rescorla, E., "Keying Material Exporters for Transport Layer Security (TLS)", RFC 5705, DOI 10.17487/RFC5705, March 2010, <https://www.rfc-editor.org/info/rfc5705>.

[RFC5756] Turner, S., Brown, D., Yiu, K., Housley, R., and T. Polk, "Updates for RSAES-OAEP and RSASSA-PSS Algorithm Parameters", RFC 5756, DOI 10.17487/RFC5756, January 2010, <https://www.rfc-editor.org/info/rfc5756>.

[RFC5869] Krawczyk, H. and P. Eronen, "HMAC-based Extract-and-Expand Key Derivation Function (HKDF)", RFC 5869, DOI 10.17487/RFC5869, May 2010, <https://www.rfc-editor.org/info/rfc5869>.

[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>.

[RFC6655] McGrew, D. and D. Bailey, "AES-CCM Cipher Suites for Transport Layer Security (TLS)", RFC 6655, DOI 10.17487/RFC6655, July 2012, <https://www.rfc-editor.org/info/rfc6655>.

[RFC6960] Santesson, S., Myers, M., Ankney, R., Malpani, A., Galperin, S., and C. Adams, "X.509 Internet Public Key Infrastructure Online Certificate Status Protocol - OCSP", RFC 6960, DOI 10.17487/RFC6960, June 2013, <https://www.rfc-editor.org/info/rfc6960>.

[RFC6961] Pettersen, Y., "The Transport Layer Security (TLS) Multiple Certificate Status Request Extension", RFC 6961, DOI 10.17487/RFC6961, June 2013, <https://www.rfc-editor.org/info/rfc6961>.

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

[RFC6979] Pornin, T., "Deterministic Usage of the Digital Signature Algorithm (DSA) and Elliptic Curve Digital Signature Algorithm (ECDSA)", RFC 6979, DOI 10.17487/RFC6979, August 2013, <https://www.rfc-editor.org/info/rfc6979>.

[RFC7301] Friedl, S., Popov, A., Langley, A., and E. Stephan, "Transport Layer Security (TLS) Application-Layer Protocol Negotiation Extension", RFC 7301, DOI 10.17487/RFC7301, July 2014, <https://www.rfc-editor.org/info/rfc7301>.

[RFC7507] Moeller, B. and A. Langley, "TLS Fallback Signaling Cipher Suite Value (SCSV) for Preventing Protocol Downgrade Attacks", RFC 7507, DOI 10.17487/RFC7507, April 2015, <https://www.rfc-editor.org/info/rfc7507>.

[RFC7539] Nir, Y. and A. Langley, "ChaCha20 and Poly1305 for IETF Protocols", RFC 7539, DOI 10.17487/RFC7539, May 2015, <https://www.rfc-editor.org/info/rfc7539>.

[RFC7748] Langley, A., Hamburg, M., and S. Turner, "Elliptic Curves for Security", RFC 7748, DOI 10.17487/RFC7748, January 2016, <https://www.rfc-editor.org/info/rfc7748>.

[RFC7919] Gillmor, D., "Negotiated Finite Field Diffie-Hellman Ephemeral Parameters for Transport Layer Security (TLS)", RFC 7919, DOI 10.17487/RFC7919, August 2016, <https://www.rfc-editor.org/info/rfc7919>.

[RFC8017] Moriarty, K., Ed., Kaliski, B., Jonsson, J., and A. Rusch, "PKCS #1: RSA Cryptography Specifications Version 2.2", RFC 8017, DOI 10.17487/RFC8017, November 2016, <https://www.rfc-editor.org/info/rfc8017>.

[RFC8032] Josefsson, S. and I. Liusvaara, "Edwards-Curve Digital Signature Algorithm (EdDSA)", RFC 8032, DOI 10.17487/RFC8032, January 2017, <https://www.rfc-editor.org/info/rfc8032>.

[RFC8126] Cotton, M., Leiba, B., and T. Narten, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 8126, DOI 10.17487/RFC8126, June 2017, <https://www.rfc-editor.org/info/rfc8126>.

[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>.

[SHS] Dang, Q., "Secure Hash Standard", National Institute of Standards and Technology report, DOI 10.6028/nist.fips.180-4, July 2015.

[X690] ITU-T, "Information technology - ASN.1 encoding Rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER)", ISO/IEC 8825-1:2002, 2002.

[X962] ANSI, "Public Key Cryptography For The Financial Services Industry: The Elliptic Curve Digital Signature Algorithm (ECDSA)", ANSI X9.62, 1998.

12.2. Informative References

[AEAD-LIMITS] Luykx, A. and K. Paterson, "Limits on Authenticated Encryption Use in TLS", 2016, <http://www.isg.rhul.ac.uk/˜kp/TLS-AEbounds.pdf>.

[Anon18] Anonymous, A., "Secure Channels for Multiplexed Data Streams: Analyzing the TLS 1.3 Record Layer Without Elision", In submission to CRYPTO 2018. RFC EDITOR: PLEASE UPDATE THIS REFERENCE AFTER FINAL NOTIFICATION (2018-4-29). , 2018.

[BBFKZG16] Bhargavan, K., Brzuska, C., Fournet, C., Kohlweiss, M., Zanella-Beguelin, S., and M. Green, "Downgrade Resilience in Key-Exchange Protocols", Proceedings of IEEE Symposium on Security and Privacy (Oakland) 2016 , 2016.

[BBK17] Bhargavan, K., Blanchet, B., and N. Kobeissi, "Verified Models and Reference Implementations for the TLS 1.3 Standard Candidate", Proceedings of IEEE Symposium on Security and Privacy (Oakland) 2017 , 2017.

[BDFKPPRSZZ16]
Bhargavan, K., Delignat-Lavaud, A., Fournet, C., Kohlweiss, M., Pan, J., Protzenko, J., Rastogi, A., Swamy, N., Zanella-Beguelin, S., and J. Zinzindohoue, "Implementing and Proving the TLS 1.3 Record Layer", Proceedings of IEEE Symposium on Security and Privacy (Oakland) 2017 , December 2016, <https://eprint.iacr.org/2016/1178>.

[Ben17a] Benjamin, D., "Presentation before the TLS WG at IETF 100", 2017, <https://datatracker.ietf.org/meeting/100/materials/slides-100-tls-sessa-tls13/>.

[Ben17b] Benjamin, D., "Additional TLS 1.3 results from Chrome", 2017, <https://www.ietf.org/mail-archive/web/tls/current/msg25168.html>.

[BMMT15] Badertscher, C., Matt, C., Maurer, U., and B. Tackmann, "Augmented Secure Channels and the Goal of the TLS 1.3 Record Layer", ProvSec 2015 , September 2015, <https://eprint.iacr.org/2015/394>.

[BT16] Bellare, M. and B. Tackmann, "The Multi-User Security of Authenticated Encryption: AES-GCM in TLS 1.3", Proceedings of CRYPTO 2016 , 2016, <https://eprint.iacr.org/2016/564>.

[CCG16] Cohn-Gordon, K., Cremers, C., and L. Garratt, "On Post-Compromise Security", IEEE Computer Security Foundations Symposium , 2015.

[CHECKOWAY]
Checkoway, S., Shacham, H., Maskiewicz, J., Garman, C., Fried, J., Cohney, S., Green, M., Heninger, N., Weinmann, R., and E. Rescorla, "A Systematic Analysis of the Juniper Dual EC Incident", Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security - CCS’16, DOI 10.1145/2976749.2978395, 2016.

[CHHSV17] Cremers, C., Horvat, M., Hoyland, J., van der Merwe, T., and S. Scott, "Awkward Handshake: Possible mismatch of client/server view on client authentication in post-handshake mode in Revision 18", 2017, <https://www.ietf.org/mail-archive/web/tls/current/msg22382.html>.

[CHSV16] Cremers, C., Horvat, M., Scott, S., and T. van der Merwe, "Automated Analysis and Verification of TLS 1.3: 0-RTT, Resumption and Delayed Authentication", Proceedings of IEEE Symposium on Security and Privacy (Oakland) 2016 , 2016, <http://ieeexplore.ieee.org/document/7546518/>.

[CK01] Canetti, R. and H. Krawczyk, "Analysis of Key-Exchange Protocols and Their Use for Building Secure Channels", Proceedings of Eurocrypt 2001 , 2001.

[CLINIC] Miller, B., Huang, L., Joseph, A., and J. Tygar, "I Know Why You Went to the Clinic: Risks and Realization of HTTPS Traffic Analysis", Privacy Enhancing Technologies pp. 143-163, DOI 10.1007/978-3-319-08506-7_8, 2014.

[DFGS15] Dowling, B., Fischlin, M., Guenther, F., and D. Stebila, "A Cryptographic Analysis of the TLS 1.3 draft-10 Full and Pre-shared Key Handshake Protocol", Proceedings of ACM CCS 2015 , 2015, <https://eprint.iacr.org/2015/914>.

[DFGS16] Dowling, B., Fischlin, M., Guenther, F., and D. Stebila, "A Cryptographic Analysis of the TLS 1.3 draft-10 Full and Pre-shared Key Handshake Protocol", TRON 2016 , 2016, <https://eprint.iacr.org/2016/081>.

[DOW92] Diffie, W., van Oorschot, P., and M. Wiener, "Authentication and authenticated key exchanges", Designs, Codes and Cryptography , 1992.

[DSS] National Institute of Standards and Technology, U.S. Department of Commerce, "Digital Signature Standard, version 4", NIST FIPS PUB 186-4, 2013.

[ECDSA] American National Standards Institute, "Public Key Cryptography for the Financial Services Industry: The Elliptic Curve Digital Signature Algorithm (ECDSA)", ANSI ANS X9.62-2005, November 2005.

[FG17] Fischlin, M. and F. Guenther, "Replay Attacks on Zero Round-Trip Time: The Case of the TLS 1.3 Handshake Candidates", Proceedings of Euro S"P 2017 , 2017, <https://eprint.iacr.org/2017/082>.

[FGSW16] Fischlin, M., Guenther, F., Schmidt, B., and B. Warinschi, "Key Confirmation in Key Exchange: A Formal Treatment and Implications for TLS 1.3", Proceedings of IEEE Symposium on Security and Privacy (Oakland) 2016 , 2016, <http://ieeexplore.ieee.org/document/7546517/>.

[FW15] Florian Weimer, ., "Factoring RSA Keys With TLS Perfect Forward Secrecy", September 2015.

[HCJ16] Husak, M., Čermak, M., Jirsik, T., and P. Čeleda, "HTTPS traffic analysis and client identification using passive SSL/TLS fingerprinting", EURASIP Journal on Information Security Vol. 2016, DOI 10.1186/s13635-016-0030-7, February 2016.

[HGFS15] Hlauschek, C., Gruber, M., Fankhauser, F., and C. Schanes, "Prying Open Pandora’s Box: KCI Attacks against TLS", Proceedings of USENIX Workshop on Offensive Technologies , 2015.

[I-D.ietf-tls-iana-registry-updates]
Salowey, J. and S. Turner, "IANA Registry Updates for TLS and DTLS", draft-ietf-tls-iana-registry-updates-04 (work in progress), February 2018.

[I-D.ietf-tls-tls13-vectors]
Thomson, M., "Example Handshake Traces for TLS 1.3", draft-ietf-tls-tls13-vectors-03 (work in progress), December 2017.

[IEEE1363]
IEEE, "Standard Specifications for Public Key Cryptography", IEEE 1363 , 2000.

[JSS15] Jager, T., Schwenk, J., and J. Somorovsky, "On the Security of TLS 1.3 and QUIC Against Weaknesses in PKCS#1 v1.5 Encryption", Proceedings of ACM CCS 2015 , 2015, <https://www.nds.rub.de/media/nds/veroeffentlichungen/2015/08/21/Tls13QuicAttacks.pdf>.

[KEYAGREEMENT]
Barker, E., Chen, L., Roginsky, A., and M. Smid, "Recommendation for Pair-Wise Key Establishment Schemes Using Discrete Logarithm Cryptography", National Institute of Standards and Technology report, DOI 10.6028/nist.sp.800-56ar2, May 2013.

[Kraw10] Krawczyk, H., "Cryptographic Extraction and Key Derivation: The HKDF Scheme", Proceedings of CRYPTO 2010 , 2010, <https://eprint.iacr.org/2010/264>.

[Kraw16] Krawczyk, H., "A Unilateral-to-Mutual Authentication Compiler for Key Exchange (with Applications to Client Authentication in TLS 1.3", Proceedings of ACM CCS 2016 , 2016, <https://eprint.iacr.org/2016/711>.

[KW16] Krawczyk, H. and H. Wee, "The OPTLS Protocol and TLS 1.3", Proceedings of Euro S"P 2016 , 2016, <https://eprint.iacr.org/2015/978>.

[LXZFH16] Li, X., Xu, J., Feng, D., Zhang, Z., and H. Hu, "Multiple Handshakes Security of TLS 1.3 Candidates", Proceedings of IEEE Symposium on Security and Privacy (Oakland) 2016 , 2016, <http://ieeexplore.ieee.org/document/7546519/>.

[Mac17] MacCarthaigh, C., "Security Review of TLS1.3 0-RTT", 2017, <https://github.com/tlswg/tls13-spec/issues/1001>.

[PSK-FINISHED]
Cremers, C., Horvat, M., van der Merwe, T., and S. Scott, "Revision 10: possible attack if client authentication is allowed during PSK", 2015, <https://www.ietf.org/mail-archive/web/tls/current/msg18215.html>.

[REKEY] Abdalla, M. and M. Bellare, "Increasing the Lifetime of a Key: A Comparative Analysis of the Security of Re-keying Techniques", ASIACRYPT2000 , October 2000.

[Res17a] Rescorla, E., "Preliminary data on Firefox TLS 1.3 Middlebox experiment", 2017, <https://www.ietf.org/mail-archive/web/tls/current/msg25091.html>.

[Res17b] Rescorla, E., "More compatibility measurement results", 2017, <https://www.ietf.org/mail-archive/web/tls/current/msg25179.html>.

[RFC3552] Rescorla, E. and B. Korver, "Guidelines for Writing RFC Text on Security Considerations", BCP 72, RFC 3552, DOI 10.17487/RFC3552, July 2003, <https://www.rfc-editor.org/info/rfc3552>.

[RFC4086] Eastlake 3rd, D., Schiller, J., and S. Crocker, "Randomness Requirements for Security", BCP 106, RFC 4086, DOI 10.17487/RFC4086, June 2005, <https://www.rfc-editor.org/info/rfc4086>.

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

[RFC4366] Blake-Wilson, S., Nystrom, M., Hopwood, D., Mikkelsen, J., and T. Wright, "Transport Layer Security (TLS) Extensions", RFC 4366, DOI 10.17487/RFC4366, April 2006, <https://www.rfc-editor.org/info/rfc4366>.

[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, <https://www.rfc-editor.org/info/rfc4492>.

[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, <https://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, <https://www.rfc-editor.org/info/rfc5246>.

[RFC5764] McGrew, D. and E. Rescorla, "Datagram Transport Layer Security (DTLS) Extension to Establish Keys for the Secure Real-time Transport Protocol (SRTP)", RFC 5764, DOI 10.17487/RFC5764, May 2010, <https://www.rfc-editor.org/info/rfc5764>.

[RFC5929] Altman, J., Williams, N., and L. Zhu, "Channel Bindings for TLS", RFC 5929, DOI 10.17487/RFC5929, July 2010, <https://www.rfc-editor.org/info/rfc5929>.

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

[RFC6176] Turner, S. and T. Polk, "Prohibiting Secure Sockets Layer (SSL) Version 2.0", RFC 6176, DOI 10.17487/RFC6176, March 2011, <https://www.rfc-editor.org/info/rfc6176>.

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

[RFC6520] Seggelmann, R., Tuexen, M., and M. Williams, "Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS) Heartbeat Extension", RFC 6520, DOI 10.17487/RFC6520, February 2012, <https://www.rfc-editor.org/info/rfc6520>.

[RFC7230] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing", RFC 7230, DOI 10.17487/RFC7230, June 2014, <https://www.rfc-editor.org/info/rfc7230>.

[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, <https://www.rfc-editor.org/info/rfc7250>.

[RFC7465] Popov, A., "Prohibiting RC4 Cipher Suites", RFC 7465, DOI 10.17487/RFC7465, February 2015, <https://www.rfc-editor.org/info/rfc7465>.

[RFC7568] Barnes, R., Thomson, M., Pironti, A., and A. Langley, "Deprecating Secure Sockets Layer Version 3.0", RFC 7568, DOI 10.17487/RFC7568, June 2015, <https://www.rfc-editor.org/info/rfc7568>.

[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, <https://www.rfc-editor.org/info/rfc7627>.

[RFC7685] Langley, A., "A Transport Layer Security (TLS) ClientHello Padding Extension", RFC 7685, DOI 10.17487/RFC7685, October 2015, <https://www.rfc-editor.org/info/rfc7685>.

[RFC7924] Santesson, S. and H. Tschofenig, "Transport Layer Security (TLS) Cached Information Extension", RFC 7924, DOI 10.17487/RFC7924, July 2016, <https://www.rfc-editor.org/info/rfc7924>.

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

[RSA] Rivest, R., Shamir, A., and L. Adleman, "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems", Communications of the ACM v. 21, n. 2, pp. 120-126., February 1978.

[SIGMA] Krawczyk, H., "SIGMA: the 'SIGn-and-MAc' approach to authenticated Diffie-Hellman and its use in the IKE protocols", Proceedings of CRYPTO 2003 , 2003.

[SLOTH] Bhargavan, K. and G. Leurent, "Transcript Collision Attacks: Breaking Authentication in TLS, IKE, and SSH", Network and Distributed System Security Symposium (NDSS 2016) , 2016.

[SSL2] Hickman, K., "The SSL Protocol", February 1995.

[SSL3] Freier, A., Karlton, P., and P. Kocher, "The SSL 3.0 Protocol", November 1996.

[TIMING] Boneh, D. and D. Brumley, "Remote timing attacks are practical", USENIX Security Symposium, 2003.

[X501] "Information Technology - Open Systems Interconnection - The Directory: Models", ITU-T X.501, 1993.

12.3. URIs

[1] mailto:tls@ietf.org

Appendix A. State Machine

This section provides a summary of the legal state transitions for the client and server handshakes. State names (in all capitals, e.g., START) have no formal meaning but are provided for ease of comprehension. Actions which are taken only in certain circumstances are indicated in []. The notation "K_{send,recv} = foo" means "set the send/recv key to the given key".

A.1. Client

```
                              START <----+
               Send ClientHello |        | Recv HelloRetryRequest
          [K_send = early data] |        |
                                v        |
           /                 WAIT_SH ----+
           |                    | Recv ServerHello
           |                    | K_recv = handshake
       Can |                    V
      send |                 WAIT_EE
     early |                    | Recv EncryptedExtensions
      data |           +--------+--------+
           |     Using |                 | Using certificate
           |       PSK |                 v
           |           |            WAIT_CERT_CR
           |           |        Recv |       | Recv CertificateRequest
           |           | Certificate |       v
           |           |             |    WAIT_CERT
           |           |             |       | Recv Certificate
           |           |             v       v
           |           |              WAIT_CV
           |           |                 | Recv CertificateVerify
           |           +> WAIT_FINISHED <+
           |                  | Recv Finished
           \                  | [Send EndOfEarlyData]
                              | K_send = handshake
                              | [Send Certificate [+ CertificateVerify]]
    Can send                  | Send Finished
    app data   -->            | K_send = K_recv = application
    after here                v
                          CONNECTED
```

Note that with the transitions as shown above, clients may send alerts that derive from post-ServerHello messages in the clear or with the early data keys. If clients need to send such alerts, they SHOULD first rekey to the handshake keys if possible.

A.2. Server

```
                              START <-----+
               Recv ClientHello |         | Send HelloRetryRequest
                                v         |
                             RECVD_CH ----+
                                | Select parameters
                                v
                             NEGOTIATED
                                | Send ServerHello
                                | K_send = handshake
                                | Send EncryptedExtensions
                                | [Send CertificateRequest]
 Can send                       | [Send Certificate + CertificateVerify]
 app data                       | Send Finished
 after   -->                    | K_send = application
 here                  +--------+--------+
              No 0-RTT |                 | 0-RTT
                       |                 |
   K_recv = handshake  |                 | K_recv = early data
 [Skip decrypt errors] |    +------> WAIT_EOED -+
                       |    |       Recv |      | Recv EndOfEarlyData
                       |    | early data |      | K_recv = handshake
                       |    +------------+      |
                       |                        |
                       +> WAIT_FLIGHT2 <--------+
                                |
                       +--------+--------+
               No auth |                 | Client auth
                       |                 |
                       |                 v
                       |             WAIT_CERT
                       |        Recv |       | Recv Certificate
                       |       empty |       v
                       | Certificate |    WAIT_CV
                       |             |       | Recv
                       |             v       | CertificateVerify
                       +-> WAIT_FINISHED <---+
                                | Recv Finished
                                | K_recv = application
                                v
                            CONNECTED
```

Appendix B. Protocol Data Structures and Constant Values

This section provides the normative protocol types and constants definitions. Values listed as _RESERVED were used in previous versions of TLS and are listed here for completeness. TLS 1.3

implementations MUST NOT send them but might receive them from older TLS implementations.

B.1. Record Layer

```
enum {
    invalid(0),
    change_cipher_spec(20),
    alert(21),
    handshake(22),
    application_data(23),
    (255)
} ContentType;

struct {
    ContentType type;
    ProtocolVersion legacy_record_version;
    uint16 length;
    opaque fragment[TLSPlaintext.length];
} TLSPlaintext;

struct {
    opaque content[TLSPlaintext.length];
    ContentType type;
    uint8 zeros[length_of_padding];
} TLSInnerPlaintext;

struct {
    ContentType opaque_type = application_data; /* 23 */
    ProtocolVersion legacy_record_version = 0x0303; /* TLS v1.2 */
    uint16 length;
    opaque encrypted_record[TLSCiphertext.length];
} TLSCiphertext;
```

B.2. Alert Messages

```
enum { warning(1), fatal(2), (255) } AlertLevel;

enum {
    close_notify(0),
    unexpected_message(10),
    bad_record_mac(20),
    decryption_failed_RESERVED(21),
    record_overflow(22),
    decompression_failure_RESERVED(30),
    handshake_failure(40),
    no_certificate_RESERVED(41),
    bad_certificate(42),
    unsupported_certificate(43),
    certificate_revoked(44),
    certificate_expired(45),
    certificate_unknown(46),
    illegal_parameter(47),
    unknown_ca(48),
    access_denied(49),
    decode_error(50),
    decrypt_error(51),
    export_restriction_RESERVED(60),
    protocol_version(70),
    insufficient_security(71),
    internal_error(80),
    inappropriate_fallback(86),
    user_canceled(90),
    no_renegotiation_RESERVED(100),
    missing_extension(109),
    unsupported_extension(110),
    certificate_unobtainable_RESERVED(111),
    unrecognized_name(112),
    bad_certificate_status_response(113),
    bad_certificate_hash_value_RESERVED(114),
    unknown_psk_identity(115),
    certificate_required(116),
    no_application_protocol(120),
    (255)
} AlertDescription;

struct {
    AlertLevel level;
    AlertDescription description;
} Alert;
```

B.3. Handshake Protocol

```
      enum {
          hello_request_RESERVED(0),
          client_hello(1),
          server_hello(2),
          hello_verify_request_RESERVED(3),
          new_session_ticket(4),
          end_of_early_data(5),
          hello_retry_request_RESERVED(6),
          encrypted_extensions(8),
          certificate(11),
          server_key_exchange_RESERVED(12),
          certificate_request(13),
          server_hello_done_RESERVED(14),
          certificate_verify(15),
          client_key_exchange_RESERVED(16),
          finished(20),
          key_update(24),
          message_hash(254),
          (255)
      } HandshakeType;

      struct {
          HandshakeType msg_type;    /* handshake type */
          uint24 length;             /* bytes in message */
          select (Handshake.msg_type) {
              case client_hello:          ClientHello;
              case server_hello:          ServerHello;
              case end_of_early_data:     EndOfEarlyData;
              case encrypted_extensions:  EncryptedExtensions;
              case certificate_request:   CertificateRequest;
              case certificate:           Certificate;
              case certificate_verify:    CertificateVerify;
              case finished:              Finished;
              case new_session_ticket:    NewSessionTicket;
              case key_update:            KeyUpdate;
          };
      } Handshake;
```

B.3.1. Key Exchange Messages

```
   uint16 ProtocolVersion;
   opaque Random[32];

   uint8 CipherSuite[2];    /* Cryptographic suite selector */

   struct {
```

```
       ProtocolVersion legacy_version = 0x0303;    /* TLS v1.2 */
       Random random;
       opaque legacy_session_id<0..32>;
       CipherSuite cipher_suites<2..2^16-2>;
       opaque legacy_compression_methods<1..2^8-1>;
       Extension extensions<8..2^16-1>;
   } ClientHello;

   struct {
       ProtocolVersion legacy_version = 0x0303;    /* TLS v1.2 */
       Random random;
       opaque legacy_session_id_echo<0..32>;
       CipherSuite cipher_suite;
       uint8 legacy_compression_method = 0;
       Extension extensions<6..2^16-1>;
   } ServerHello;

   struct {
       ExtensionType extension_type;
       opaque extension_data<0..2^16-1>;
   } Extension;

   enum {
       server_name(0),                             /* RFC 6066 */
       max_fragment_length(1),                     /* RFC 6066 */
       status_request(5),                          /* RFC 6066 */
       supported_groups(10),                       /* RFC 4492, 7919 */
       signature_algorithms(13),                   /* [[this document]] */
       use_srtp(14),                               /* RFC 5764 */
       heartbeat(15),                              /* RFC 6520 */
       application_layer_protocol_negotiation(16), /* RFC 7301 */
       signed_certificate_timestamp(18),           /* RFC 6962 */
       client_certificate_type(19),                /* RFC 7250 */
       server_certificate_type(20),                /* RFC 7250 */
       padding(21),                                /* RFC 7685 */
       RESERVED(40),                               /* Used but never assigned */
       pre_shared_key(41),                         /* [[this document]] */
       early_data(42),                             /* [[this document]] */
       supported_versions(43),                     /* [[this document]] */
       cookie(44),                                 /* [[this document]] */
       psk_key_exchange_modes(45),                 /* [[this document]] */
       RESERVED(46),                               /* Used but never assigned */
       certificate_authorities(47),                /* [[this document]] */
       oid_filters(48),                            /* [[this document]] */
       post_handshake_auth(49),                    /* [[this document]] */
       signature_algorithms_cert(50),              /* [[this document]] */
       key_share(51),                              /* [[this document]] */
       (65535)
```

```
   } ExtensionType;

   struct {
       NamedGroup group;
       opaque key_exchange<1..2^16-1>;
   } KeyShareEntry;

   struct {
       KeyShareEntry client_shares<0..2^16-1>;
   } KeyShareClientHello;

   struct {
       NamedGroup selected_group;
   } KeyShareHelloRetryRequest;

   struct {
       KeyShareEntry server_share;
   } KeyShareServerHello;

   struct {
       uint8 legacy_form = 4;
       opaque X[coordinate_length];
       opaque Y[coordinate_length];
   } UncompressedPointRepresentation;

   enum { psk_ke(0), psk_dhe_ke(1), (255) } PskKeyExchangeMode;

   struct {
       PskKeyExchangeMode ke_modes<1..255>;
   } PskKeyExchangeModes;

   struct {} Empty;

   struct {
       select (Handshake.msg_type) {
           case new_session_ticket:   uint32 max_early_data_size;
           case client_hello:         Empty;
           case encrypted_extensions: Empty;
       };
   } EarlyDataIndication;

   struct {
       opaque identity<1..2^16-1>;
       uint32 obfuscated_ticket_age;
   } PskIdentity;

   opaque PskBinderEntry<32..255>;
```

```
   struct {
       PskIdentity identities<7..2^16-1>;
       PskBinderEntry binders<33..2^16-1>;
   } OfferedPsks;

   struct {
       select (Handshake.msg_type) {
           case client_hello: OfferedPsks;
           case server_hello: uint16 selected_identity;
       };
   } PreSharedKeyExtension;
```

B.3.1.1. Version Extension

```
      struct {
          select (Handshake.msg_type) {
              case client_hello:
                   ProtocolVersion versions<2..254>;

              case server_hello: /* and HelloRetryRequest */
                   ProtocolVersion selected_version;
          };
      } SupportedVersions;
```

B.3.1.2. Cookie Extension

```
      struct {
          opaque cookie<1..2^16-1>;
      } Cookie;
```

B.3.1.3. Signature Algorithm Extension

```
enum {
    /* RSASSA-PKCS1-v1_5 algorithms */
    rsa_pkcs1_sha256(0x0401),
    rsa_pkcs1_sha384(0x0501),
    rsa_pkcs1_sha512(0x0601),

    /* ECDSA algorithms */
    ecdsa_secp256r1_sha256(0x0403),
    ecdsa_secp384r1_sha384(0x0503),
    ecdsa_secp521r1_sha512(0x0603),

    /* RSASSA-PSS algorithms with public key OID rsaEncryption */
    rsa_pss_rsae_sha256(0x0804),
    rsa_pss_rsae_sha384(0x0805),
    rsa_pss_rsae_sha512(0x0806),

    /* EdDSA algorithms */
    ed25519(0x0807),
    ed448(0x0808),

    /* RSASSA-PSS algorithms with public key OID RSASSA-PSS */
    rsa_pss_pss_sha256(0x0809),
    rsa_pss_pss_sha384(0x080a),
    rsa_pss_pss_sha512(0x080b),

    /* Legacy algorithms */
    rsa_pkcs1_sha1(0x0201),
    ecdsa_sha1(0x0203),

    /* Reserved Code Points */
    obsolete_RESERVED(0x0000..0x0200),
    dsa_sha1_RESERVED(0x0202),
    obsolete_RESERVED(0x0204..0x0400),
    dsa_sha256_RESERVED(0x0402),
    obsolete_RESERVED(0x0404..0x0500),
    dsa_sha384_RESERVED(0x0502),
    obsolete_RESERVED(0x0504..0x0600),
    dsa_sha512_RESERVED(0x0602),
    obsolete_RESERVED(0x0604..0x06FF),
    private_use(0xFE00..0xFFFF),
    (0xFFFF)
} SignatureScheme;

struct {
    SignatureScheme supported_signature_algorithms<2..2^16-2>;
} SignatureSchemeList;
```

B.3.1.4. Supported Groups Extension

```
enum {
    unallocated_RESERVED(0x0000),

    /* Elliptic Curve Groups (ECDHE) */
    obsolete_RESERVED(0x0001..0x0016),
    secp256r1(0x0017), secp384r1(0x0018), secp521r1(0x0019),
    obsolete_RESERVED(0x001A..0x001C),
    x25519(0x001D), x448(0x001E),

    /* Finite Field Groups (DHE) */
    ffdhe2048(0x0100), ffdhe3072(0x0101), ffdhe4096(0x0102),
    ffdhe6144(0x0103), ffdhe8192(0x0104),

    /* Reserved Code Points */
    ffdhe_private_use(0x01FC..0x01FF),
    ecdhe_private_use(0xFE00..0xFEFF),
    obsolete_RESERVED(0xFF01..0xFF02),
    (0xFFFF)
} NamedGroup;

struct {
    NamedGroup named_group_list<2..2^16-1>;
} NamedGroupList;
```

Values within "obsolete_RESERVED" ranges are used in previous versions of TLS and MUST NOT be offered or negotiated by TLS 1.3 implementations. The obsolete curves have various known/theoretical weaknesses or have had very little usage, in some cases only due to unintentional server configuration issues. They are no longer considered appropriate for general use and should be assumed to be potentially unsafe. The set of curves specified here is sufficient for interoperability with all currently deployed and properly configured TLS implementations.

B.3.2. Server Parameters Messages

```
opaque DistinguishedName<1..2^16-1>;

struct {
    DistinguishedName authorities<3..2^16-1>;
} CertificateAuthoritiesExtension;

struct {
    opaque certificate_extension_oid<1..2^8-1>;
    opaque certificate_extension_values<0..2^16-1>;
} OIDFilter;

struct {
    OIDFilter filters<0..2^16-1>;
} OIDFilterExtension;

struct {} PostHandshakeAuth;

struct {
    Extension extensions<0..2^16-1>;
} EncryptedExtensions;

struct {
    opaque certificate_request_context<0..2^8-1>;
    Extension extensions<2..2^16-1>;
} CertificateRequest;
```

B.3.3. Authentication Messages

```
/* Managed by IANA */
enum {
    X509(0),
    OpenPGP_RESERVED(1),
    RawPublicKey(2),
    (255)
} CertificateType;

struct {
    select (certificate_type) {
        case RawPublicKey:
          /* From RFC 7250 ASN.1_subjectPublicKeyInfo */
          opaque ASN1_subjectPublicKeyInfo<1..2^24-1>;

        case X509:
          opaque cert_data<1..2^24-1>;
    };
    Extension extensions<0..2^16-1>;
} CertificateEntry;

struct {
    opaque certificate_request_context<0..2^8-1>;
    CertificateEntry certificate_list<0..2^24-1>;
} Certificate;

struct {
    SignatureScheme algorithm;
    opaque signature<0..2^16-1>;
} CertificateVerify;

struct {
    opaque verify_data[Hash.length];
} Finished;
```

B.3.4. Ticket Establishment

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

B.3.5. Updating Keys

```
struct {} EndOfEarlyData;

enum {
    update_not_requested(0), update_requested(1), (255)
} KeyUpdateRequest;

struct {
    KeyUpdateRequest request_update;
} KeyUpdate;
```

B.4. Cipher Suites

A symmetric cipher suite defines the pair of the AEAD algorithm and hash algorithm to be used with HKDF. Cipher suite names follow the naming convention:

```
CipherSuite TLS_AEAD_HASH = VALUE;
```

Component	Contents
TLS	The string "TLS"
AEAD	The AEAD algorithm used for record protection
HASH	The hash algorithm used with HKDF
VALUE	The two byte ID assigned for this cipher suite

This specification defines the following cipher suites for use with TLS 1.3.

Description	Value
TLS_AES_128_GCM_SHA256	{0x13,0x01}
TLS_AES_256_GCM_SHA384	{0x13,0x02}
TLS_CHACHA20_POLY1305_SHA256	{0x13,0x03}
TLS_AES_128_CCM_SHA256	{0x13,0x04}
TLS_AES_128_CCM_8_SHA256	{0x13,0x05}

The corresponding AEAD algorithms AEAD_AES_128_GCM, AEAD_AES_256_GCM, and AEAD_AES_128_CCM are defined in [RFC5116]. AEAD_CHACHA20_POLY1305 is defined in [RFC7539]. AEAD_AES_128_CCM_8 is defined in [RFC6655]. The corresponding hash algorithms are defined in [SHS].

Although TLS 1.3 uses the same cipher suite space as previous versions of TLS, TLS 1.3 cipher suites are defined differently, only specifying the symmetric ciphers, and cannot be used for TLS 1.2. Similarly, TLS 1.2 and lower cipher suites cannot be used with TLS 1.3.

New cipher suite values are assigned by IANA as described in Section 11.

Appendix C. Implementation Notes

The TLS protocol cannot prevent many common security mistakes. This section provides several recommendations to assist implementors. [I-D.ietf-tls-tls13-vectors] provides test vectors for TLS 1.3 handshakes.

C.1. Random Number Generation and Seeding

TLS requires a cryptographically secure pseudorandom number generator (CSPRNG). In most cases, the operating system provides an appropriate facility such as /dev/urandom, which should be used absent other (performance) concerns. It is RECOMMENDED to use an existing CSPRNG implementation in preference to crafting a new one. Many adequate cryptographic libraries are already available under favorable license terms. Should those prove unsatisfactory, [RFC4086] provides guidance on the generation of random values.

TLS uses random values both in public protocol fields such as the public Random values in the ClientHello and ServerHello and to generate keying material. With a properly functioning CSPRNG, this does not present a security problem as it is not feasible to determine the CSPRNG state from its output. However, with a broken CSPRNG, it may be possible for an attacker to use the public output to determine the CSPRNG internal state and thereby predict the keying material, as documented in [CHECKOWAY]. Implementations can provide extra security against this form of attack by using separate CSPRNGs to generate public and private values.

C.2. Certificates and Authentication

Implementations are responsible for verifying the integrity of certificates and should generally support certificate revocation messages. Absent a specific indication from an application profile, Certificates should always be verified to ensure proper signing by a trusted Certificate Authority (CA). The selection and addition of trust anchors should be done very carefully. Users should be able to view information about the certificate and trust anchor. Applications SHOULD also enforce minimum and maximum key sizes. For example, certification paths containing keys or signatures weaker than 2048-bit RSA or 224-bit ECDSA are not appropriate for secure applications.

C.3. Implementation Pitfalls

Implementation experience has shown that certain parts of earlier TLS specifications are not easy to understand and have been a source of interoperability and security problems. Many of these areas have been clarified in this document but this appendix contains a short list of the most important things that require special attention from implementors.

TLS protocol issues:

- Do you correctly handle handshake messages that are fragmented to multiple TLS records (see Section 5.1)? Including corner cases like a ClientHello that is split to several small fragments? Do you fragment handshake messages that exceed the maximum fragment size? In particular, the Certificate and CertificateRequest handshake messages can be large enough to require fragmentation.

- Do you ignore the TLS record layer version number in all unencrypted TLS records? (see Appendix D)

- Have you ensured that all support for SSL, RC4, EXPORT ciphers, and MD5 (via the "signature_algorithms" extension) is completely removed from all possible configurations that support TLS 1.3 or later, and that attempts to use these obsolete capabilities fail correctly? (see Appendix D)

- Do you handle TLS extensions in ClientHello correctly, including unknown extensions?

- When the server has requested a client certificate, but no suitable certificate is available, do you correctly send an empty Certificate message, instead of omitting the whole message (see Section 4.4.2.3)?

- When processing the plaintext fragment produced by AEAD-Decrypt and scanning from the end for the ContentType, do you avoid scanning past the start of the cleartext in the event that the peer has sent a malformed plaintext of all-zeros?

- Do you properly ignore unrecognized cipher suites (Section 4.1.2), hello extensions (Section 4.2), named groups (Section 4.2.7), key shares (Section 4.2.8), supported versions (Section 4.2.1), and signature algorithms (Section 4.2.3) in the ClientHello?

- As a server, do you send a HelloRetryRequest to clients which support a compatible (EC)DHE group but do not predict it in the "key_share" extension? As a client, do you correctly handle a HelloRetryRequest from the server?

Cryptographic details:

- What countermeasures do you use to prevent timing attacks [TIMING]?

- When using Diffie-Hellman key exchange, do you correctly preserve leading zero bytes in the negotiated key (see Section 7.4.1)?

- Does your TLS client check that the Diffie-Hellman parameters sent by the server are acceptable, (see Section 4.2.8.1)?

- Do you use a strong and, most importantly, properly seeded random number generator (see Appendix C.1) when generating Diffie-Hellman private values, the ECDSA "k" parameter, and other security-critical values? It is RECOMMENDED that implementations implement "deterministic ECDSA" as specified in [RFC6979].

- Do you zero-pad Diffie-Hellman public key values to the group size (see Section 4.2.8.1)?

- Do you verify signatures after making them to protect against RSA-CRT key leaks? [FW15]

C.4. Client Tracking Prevention

Clients SHOULD NOT reuse a ticket for multiple connections. Reuse of a ticket allows passive observers to correlate different connections. Servers that issue tickets SHOULD offer at least as many tickets as the number of connections that a client might use; for example, a web browser using HTTP/1.1 [RFC7230] might open six connections to a server. Servers SHOULD issue new tickets with every connection. This ensures that clients are always able to use a new ticket when creating a new connection.

C.5. Unauthenticated Operation

Previous versions of TLS offered explicitly unauthenticated cipher suites based on anonymous Diffie-Hellman. These modes have been deprecated in TLS 1.3. However, it is still possible to negotiate parameters that do not provide verifiable server authentication by several methods, including:

- Raw public keys [RFC7250].
- Using a public key contained in a certificate but without validation of the certificate chain or any of its contents.

Either technique used alone is vulnerable to man-in-the-middle attacks and therefore unsafe for general use. However, it is also possible to bind such connections to an external authentication mechanism via out-of-band validation of the server's public key, trust on first use, or a mechanism such as channel bindings (though the channel bindings described in [RFC5929] are not defined for TLS 1.3). If no such mechanism is used, then the connection has no protection against active man-in-the-middle attack; applications MUST NOT use TLS in such a way absent explicit configuration or a specific application profile.

Appendix D. Backward Compatibility

The TLS protocol provides a built-in mechanism for version negotiation between endpoints potentially supporting different versions of TLS.

TLS 1.x and SSL 3.0 use compatible ClientHello messages. Servers can also handle clients trying to use future versions of TLS as long as the ClientHello format remains compatible and there is at least one protocol version supported by both the client and the server.

Prior versions of TLS used the record layer version number (TLSPlaintext.legacy_record_version and TLSCiphertext.legacy_record_version) for various purposes. As of TLS 1.3, this field is deprecated. The value of TLSPlaintext.legacy_record_version MUST be ignored by all implementations. The value of TLSCiphertext.legacy_record_version is included in the additional data for deprotection but MAY otherwise be ignored or MAY be validated to match the fixed constant value. Version negotiation is performed using only the handshake versions (ClientHello.legacy_version, ServerHello.legacy_version, as well as the ClientHello, HelloRetryRequest and ServerHello "supported_versions" extensions). In order to maximize interoperability with older endpoints, implementations that negotiate the use of TLS 1.0-1.2 SHOULD set the record layer version number to the negotiated version for the ServerHello and all records thereafter.

For maximum compatibility with previously non-standard behavior and misconfigured deployments, all implementations SHOULD support validation of certification paths based on the expectations in this document, even when handling prior TLS versions' handshakes. (see Section 4.4.2.2)

TLS 1.2 and prior supported an "Extended Master Secret" [RFC7627] extension which digested large parts of the handshake transcript into the master secret. Because TLS 1.3 always hashes in the transcript up to the server CertificateVerify, implementations which support both TLS 1.3 and earlier versions SHOULD indicate the use of the Extended Master Secret extension in their APIs whenever TLS 1.3 is used.

D.1. Negotiating with an older server

A TLS 1.3 client who wishes to negotiate with servers that do not support TLS 1.3 will send a normal TLS 1.3 ClientHello containing 0x0303 (TLS 1.2) in ClientHello.legacy_version but with the correct version(s) in the "supported_versions" extension. If the server does not support TLS 1.3 it will respond with a ServerHello containing an older version number. If the client agrees to use this version, the negotiation will proceed as appropriate for the negotiated protocol. A client using a ticket for resumption SHOULD initiate the connection using the version that was previously negotiated.

Note that 0-RTT data is not compatible with older servers and SHOULD NOT be sent absent knowledge that the server supports TLS 1.3. See Appendix D.3.

If the version chosen by the server is not supported by the client (or not acceptable), the client MUST abort the handshake with a "protocol_version" alert.

Some legacy server implementations are known to not implement the TLS specification properly and might abort connections upon encountering TLS extensions or versions which they are not aware of. Interoperability with buggy servers is a complex topic beyond the scope of this document. Multiple connection attempts may be required in order to negotiate a backwards compatible connection; however, this practice is vulnerable to downgrade attacks and is NOT RECOMMENDED.

D.2. Negotiating with an older client

A TLS server can also receive a ClientHello indicating a version number smaller than its highest supported version. If the "supported_versions" extension is present, the server MUST negotiate using that extension as described in Section 4.2.1. If the "supported_versions" extension is not present, the server MUST negotiate the minimum of ClientHello.legacy_version and TLS 1.2. For example, if the server supports TLS 1.0, 1.1, and 1.2, and legacy_version is TLS 1.0, the server will proceed with a TLS 1.0 ServerHello. If the "supported_versions" extension is absent and the server only supports versions greater than ClientHello.legacy_version, the server MUST abort the handshake with a "protocol_version" alert.

Note that earlier versions of TLS did not clearly specify the record layer version number value in all cases (TLSPlaintext.legacy_record_version). Servers will receive various TLS 1.x versions in this field, but its value MUST always be ignored.

D.3. 0-RTT backwards compatibility

0-RTT data is not compatible with older servers. An older server will respond to the ClientHello with an older ServerHello, but it will not correctly skip the 0-RTT data and will fail to complete the handshake. This can cause issues when a client attempts to use 0-RTT, particularly against multi-server deployments. For example, a deployment could deploy TLS 1.3 gradually with some servers implementing TLS 1.3 and some implementing TLS 1.2, or a TLS 1.3 deployment could be downgraded to TLS 1.2.

A client that attempts to send 0-RTT data MUST fail a connection if it receives a ServerHello with TLS 1.2 or older. A client that attempts to repair this error SHOULD NOT send a TLS 1.2 ClientHello, but instead send a TLS 1.3 ClientHello without 0-RTT data.

To avoid this error condition, multi-server deployments SHOULD ensure a uniform and stable deployment of TLS 1.3 without 0-RTT prior to enabling 0-RTT.

D.4. Middlebox Compatibility Mode

Field measurements [Ben17a], [Ben17b], [Res17a], [Res17b] have found that a significant number of middleboxes misbehave when a TLS client/server pair negotiates TLS 1.3. Implementations can increase the chance of making connections through those middleboxes by making the TLS 1.3 handshake look more like a TLS 1.2 handshake:

- The client always provides a non-empty session ID in the ClientHello, as described in the legacy_session_id section of Section 4.1.2.

- If not offering early data, the client sends a dummy change_cipher_spec record (see the third paragraph of Section 5.1) immediately before its second flight. This may either be before its second ClientHello or before its encrypted handshake flight. If offering early data, the record is placed immediately after the first ClientHello.

- The server sends a dummy change_cipher_spec record immediately after its first handshake message. This may either be after a ServerHello or a HelloRetryRequest.

When put together, these changes make the TLS 1.3 handshake resemble TLS 1.2 session resumption, which improves the chance of successfully connecting through middleboxes. This "compatibility mode" is partially negotiated: The client can opt to provide a session ID or not and the server has to echo it. Either side can send change_cipher_spec at any time during the handshake, as they must be ignored by the peer, but if the client sends a non-empty session ID, the server MUST send the change_cipher_spec as described in this section.

D.5. Backwards Compatibility Security Restrictions

Implementations negotiating use of older versions of TLS SHOULD prefer forward secret and AEAD cipher suites, when available.

The security of RC4 cipher suites is considered insufficient for the reasons cited in [RFC7465]. Implementations MUST NOT offer or negotiate RC4 cipher suites for any version of TLS for any reason.

Old versions of TLS permitted the use of very low strength ciphers. Ciphers with a strength less than 112 bits MUST NOT be offered or negotiated for any version of TLS for any reason.

The security of SSL 3.0 [SSL3] is considered insufficient for the reasons enumerated in [RFC7568], and it MUST NOT be negotiated for any reason.

The security of SSL 2.0 [SSL2] is considered insufficient for the reasons enumerated in [RFC6176], and it MUST NOT be negotiated for any reason.

Implementations MUST NOT send an SSL version 2.0 compatible CLIENT-HELLO. Implementations MUST NOT negotiate TLS 1.3 or later using an

SSL version 2.0 compatible CLIENT-HELLO. Implementations are NOT RECOMMENDED to accept an SSL version 2.0 compatible CLIENT-HELLO in order to negotiate older versions of TLS.

Implementations MUST NOT send a ClientHello.legacy_version or ServerHello.legacy_version set to 0x0300 or less. Any endpoint receiving a Hello message with ClientHello.legacy_version or ServerHello.legacy_version set to 0x0300 MUST abort the handshake with a "protocol_version" alert.

Implementations MUST NOT send any records with a version less than 0x0300. Implementations SHOULD NOT accept any records with a version less than 0x0300 (but may inadvertently do so if the record version number is ignored completely).

Implementations MUST NOT use the Truncated HMAC extension, defined in Section 7 of [RFC6066], as it is not applicable to AEAD algorithms and has been shown to be insecure in some scenarios.

Appendix E. Overview of Security Properties

A complete security analysis of TLS is outside the scope of this document. In this section, we provide an informal description the desired properties as well as references to more detailed work in the research literature which provides more formal definitions.

We cover properties of the handshake separately from those of the record layer.

E.1. Handshake

The TLS handshake is an Authenticated Key Exchange (AKE) protocol which is intended to provide both one-way authenticated (server-only) and mutually authenticated (client and server) functionality. At the completion of the handshake, each side outputs its view of the following values:

- A set of "session keys" (the various secrets derived from the master secret) from which can be derived a set of working keys.

- A set of cryptographic parameters (algorithms, etc.)

- The identities of the communicating parties.

We assume the attacker to be an active network attacker, which means it has complete control over the network used to communicate between the parties [RFC3552]. Even under these conditions, the handshake should provide the properties listed below. Note that these properties are not necessarily independent, but reflect the protocol consumers' needs.

Establishing the same session keys. The handshake needs to output the same set of session keys on both sides of the handshake, provided that it completes successfully on each endpoint (See [CK01]; defn 1, part 1).

Secrecy of the session keys. The shared session keys should be known only to the communicating parties and not to the attacker (See [CK01]; defn 1, part 2). Note that in a unilaterally authenticated connection, the attacker can establish its own session keys with the server, but those session keys are distinct from those established by the client.

Peer Authentication. The client's view of the peer identity should reflect the server's identity. If the client is authenticated, the server's view of the peer identity should match the client's identity.

Uniqueness of the session keys: Any two distinct handshakes should produce distinct, unrelated session keys. Individual session keys produced by a handshake should also be distinct and independent.

Downgrade protection. The cryptographic parameters should be the same on both sides and should be the same as if the peers had been communicating in the absence of an attack (See [BBFKZG16]; defns 8 and 9}).

Forward secret with respect to long-term keys If the long-term keying material (in this case the signature keys in certificate-based authentication modes or the external/resumption PSK in PSK with (EC)DHE modes) is compromised after the handshake is complete, this does not compromise the security of the session key (See [DOW92]), as long as the session key itself has been erased. The forward secrecy property is not satisfied when PSK is used in the "psk_ke" PskKeyExchangeMode.

Key Compromise Impersonation (KCI) resistance In a mutually-authenticated connection with certificates, compromising the long-term secret of one actor should not break that actor's authentication of their peer in the given connection (see [HGFS15]). For example, if a client's signature key is compromised, it should not be possible to impersonate arbitrary servers to that client in subsequent handshakes.

Protection of endpoint identities. The server's identity (certificate) should be protected against passive attackers. The client's identity should be protected against both passive and active attackers.

Informally, the signature-based modes of TLS 1.3 provide for the establishment of a unique, secret, shared key established by an (EC)DHE key exchange and authenticated by the server's signature over the handshake transcript, as well as tied to the server's identity by a MAC. If the client is authenticated by a certificate, it also signs over the handshake transcript and provides a MAC tied to both identities. [SIGMA] describes the design and analysis of this type of key exchange protocol. If fresh (EC)DHE keys are used for each connection, then the output keys are forward secret.

The external PSK and resumption PSK bootstrap from a long-term shared secret into a unique per-connection set of short-term session keys. This secret may have been established in a previous handshake. If PSK with (EC)DHE key establishment is used, these session keys will also be forward secret. The resumption PSK has been designed so that the resumption master secret computed by connection N and needed to form connection N+1 is separate from the traffic keys used by connection N, thus providing forward secrecy between the connections. In addition, if multiple tickets are established on the same connection, they are associated with different keys, so compromise of the PSK associated with one ticket does not lead to the compromise of connections established with PSKs associated with other tickets. This property is most interesting if tickets are stored in a database (and so can be deleted) rather than if they are self-encrypted.

The PSK binder value forms a binding between a PSK and the current handshake, as well as between the session where the PSK was established and the current session. This binding transitively includes the original handshake transcript, because that transcript is digested into the values which produce the Resumption Master Secret. This requires that both the KDF used to produce the resumption master secret and the MAC used to compute the binder be collision resistant. See Appendix E.1.1 for more on this. Note: The binder does not cover the binder values from other PSKs, though they are included in the Finished MAC.

Note: TLS does not currently permit the server to send a certificate_request message in non-certificate-based handshakes (e.g., PSK). If this restriction were to be relaxed in future, the client's signature would not cover the server's certificate directly. However, if the PSK was established through a NewSessionTicket, the client's signature would transitively cover the server's certificate through the PSK binder. [PSK-FINISHED] describes a concrete attack on constructions that do not bind to the server's certificate (see also [Kraw16]). It is unsafe to use certificate-based client authentication when the client might potentially share the same PSK/key-id pair with two different endpoints. Implementations MUST NOT combine external PSKs with certificate-based authentication of either the client or the server unless negotiated by some extension.

If an exporter is used, then it produces values which are unique and secret (because they are generated from a unique session key). Exporters computed with different labels and contexts are computationally independent, so it is not feasible to compute one from another or the session secret from the exported value. Note: exporters can produce arbitrary-length values. If exporters are to be used as channel bindings, the exported value MUST be large enough to provide collision resistance. The exporters provided in TLS 1.3 are derived from the same handshake contexts as the early traffic keys and the application traffic keys respectively, and thus have similar security properties. Note that they do not include the client's certificate; future applications which wish to bind to the client's certificate may need to define a new exporter that includes the full handshake transcript.

For all handshake modes, the Finished MAC (and where present, the signature), prevents downgrade attacks. In addition, the use of certain bytes in the random nonces as described in Section 4.1.3 allows the detection of downgrade to previous TLS versions. See [BBFKZG16] for more detail on TLS 1.3 and downgrade.

As soon as the client and the server have exchanged enough information to establish shared keys, the remainder of the handshake is encrypted, thus providing protection against passive attackers, even if the computed shared key is not authenticated. Because the server authenticates before the client, the client can ensure that if it authenticates to the server, it only reveals its identity to an authenticated server. Note that implementations must use the provided record padding mechanism during the handshake to avoid leaking information about the identities due to length. The client's proposed PSK identities are not encrypted, nor is the one that the server selects.

E.1.1. Key Derivation and HKDF

Key derivation in TLS 1.3 uses the HKDF function defined in [RFC5869] and its two components, HKDF-Extract and HKDF-Expand. The full rationale for the HKDF construction can be found in [Kraw10] and the rationale for the way it is used in TLS 1.3 in [KW16]. Throughout this document, each application of HKDF-Extract is followed by one or more invocations of HKDF-Expand. This ordering should always be followed (including in future revisions of this document), in particular, one SHOULD NOT use an output of HKDF-Extract as an input to another application of HKDF-Extract without an HKDF-Expand in between. Consecutive applications of HKDF-Expand are allowed as long as these are differentiated via the key and/or the labels.

Note that HKDF-Expand implements a pseudorandom function (PRF) with both inputs and outputs of variable length. In some of the uses of HKDF in this document (e.g., for generating exporters and the resumption_master_secret), it is necessary that the application of HKDF-Expand be collision-resistant, namely, it should be infeasible to find two different inputs to HKDF-Expand that output the same value. This requires the underlying hash function to be collision resistant and the output length from HKDF-Expand to be of size at least 256 bits (or as much as needed for the hash function to prevent finding collisions).

E.1.2. Client Authentication

A client that has sent authentication data to a server, either during the handshake or in post-handshake authentication, cannot be sure if the server afterwards considers the client to be authenticated or not. If the client needs to determine if the server considers the connection to be unilaterally or mutually authenticated, this has to be provisioned by the application layer. See [CHHSV17] for details. In addition, the analysis of post-handshake authentication from [Kraw16] shows that the client identified by the certificate sent in the post-handshake phase possesses the traffic key. This party is therefore the client that participated in the original handshake or one to whom the original client delegated the traffic key (assuming that the traffic key has not been compromised).

E.1.3. 0-RTT

The 0-RTT mode of operation generally provides similar security properties as 1-RTT data, with the two exceptions that the 0-RTT encryption keys do not provide full forward secrecy and that the server is not able to guarantee uniqueness of the handshake (non-replayability) without keeping potentially undue amounts of state. See Section 8 for mechanisms to limit the exposure to replay.

E.1.4. Exporter Independence

The exporter_master_secret and early_exporter_master_secret are derived to be independent of the traffic keys and therefore do not represent a threat to the security of traffic encrypted with those keys. However, because these secrets can be used to compute any exporter value, they SHOULD be erased as soon as possible. If the total set of exporter labels is known, then implementations SHOULD pre-compute the inner Derive-Secret stage of the exporter computation for all those labels, then erase the [early_]exporter_master_secret, followed by each inner values as soon as it is known that it will not be needed again.

E.1.5. Post-Compromise Security

TLS does not provide security for handshakes which take place after the peer's long-term secret (signature key or external PSK) is compromised. It therefore does not provide post-compromise security [CCG16], sometimes also referred to as backwards or future secrecy. This is in contrast to KCI resistance, which describes the security guarantees that a party has after its own long-term secret has been compromised.

E.1.6. External References

The reader should refer to the following references for analysis of the TLS handshake: [DFGS15] [CHSV16] [DFGS16] [KW16] [Kraw16] [FGSW16] [LXZFH16] [FG17] [BBK17].

E.2. Record Layer

The record layer depends on the handshake producing strong traffic secrets which can be used to derive bidirectional encryption keys and nonces. Assuming that is true, and the keys are used for no more data than indicated in Section 5.5 then the record layer should provide the following guarantees:

Confidentiality. An attacker should not be able to determine the plaintext contents of a given record.

Integrity. An attacker should not be able to craft a new record which is different from an existing record which will be accepted by the receiver.

Order protection/non-replayability An attacker should not be able to cause the receiver to accept a record which it has already accepted or cause the receiver to accept record N+1 without having first processed record N.

Length concealment. Given a record with a given external length, the attacker should not be able to determine the amount of the record that is content versus padding.

Forward secrecy after key change. If the traffic key update mechanism described in Section 4.6.3 has been used and the previous generation key is deleted, an attacker who compromises the endpoint should not be able to decrypt traffic encrypted with the old key.

Informally, TLS 1.3 provides these properties by AEAD-protecting the plaintext with a strong key. AEAD encryption [RFC5116] provides confidentiality and integrity for the data. Non-replayability is provided by using a separate nonce for each record, with the nonce being derived from the record sequence number (Section 5.3), with the sequence number being maintained independently at both sides thus records which are delivered out of order result in AEAD deprotection failures. In order to prevent mass cryptanalysis when the same plaintext is repeatedly encrypted by different users under the same key (as is commonly the case for HTTP), the nonce is formed by mixing the sequence number with a secret per-connection initialization vector derived along with the traffic keys. See [BT16] for analysis of this construction.

The re-keying technique in TLS 1.3 (see Section 7.2) follows the construction of the serial generator in [REKEY], which shows that re-keying can allow keys to be used for a larger number of encryptions than without re-keying. This relies on the security of the HKDF-Expand-Label function as a pseudorandom function (PRF). In addition, as long as this function is truly one way, it is not possible to compute traffic keys from prior to a key change (forward secrecy).

TLS does not provide security for data which is communicated on a connection after a traffic secret of that connection is compromised. That is, TLS does not provide post-compromise security/future secrecy/backward secrecy with respect to the traffic secret. Indeed, an attacker who learns a traffic secret can compute all future traffic secrets on that connection. Systems which want such guarantees need to do a fresh handshake and establish a new connection with an (EC)DHE exchange.

E.2.1. External References

The reader should refer to the following references for analysis of the TLS record layer: [BMMT15] [BT16] [BDFKPPRSZZ16] [BBK17] [Anon18].

E.3. Traffic Analysis

TLS is susceptible to a variety of traffic analysis attacks based on observing the length and timing of encrypted packets [CLINIC] [HCJ16]. This is particularly easy when there is a small set of possible messages to be distinguished, such as for a video server hosting a fixed corpus of content, but still provides usable information even in more complicated scenarios.

TLS does not provide any specific defenses against this form of attack but does include a padding mechanism for use by applications: The plaintext protected by the AEAD function consists of content plus variable-length padding, which allows the application to produce arbitrary length encrypted records as well as padding-only cover traffic to conceal the difference between periods of transmission and periods of silence. Because the padding is encrypted alongside the actual content, an attacker cannot directly determine the length of the padding, but may be able to measure it indirectly by the use of timing channels exposed during record processing (i.e., seeing how long it takes to process a record or trickling in records to see which ones elicit a response from the server). In general, it is not known how to remove all of these channels because even a constant time padding removal function will likely feed the content into data-dependent functions. At minimum, a fully constant time server or client would require close cooperation with the application layer protocol implementation, including making that higher level protocol constant time.

Note: Robust traffic analysis defences will likely lead to inferior performance due to delay in transmitting packets and increased traffic volume.

E.4. Side Channel Attacks

In general, TLS does not have specific defenses against side-channel attacks (i.e., those which attack the communications via secondary channels such as timing) leaving those to the implementation of the relevant cryptographic primitives. However, certain features of TLS are designed to make it easier to write side-channel resistant code:

- Unlike previous versions of TLS which used a composite MAC-then-encrypt structure, TLS 1.3 only uses AEAD algorithms, allowing implementations to use self-contained constant-time implementations of those primitives.

- TLS uses a uniform "bad_record_mac" alert for all decryption errors, which is intended to prevent an attacker from gaining piecewise insight into portions of the message. Additional resistance is provided by terminating the connection on such errors; a new connection will have different cryptographic material, preventing attacks against the cryptographic primitives that require multiple trials.

Information leakage through side channels can occur at layers above TLS, in application protocols and the applications that use them. Resistance to side-channel attacks depends on applications and application protocols separately ensuring that confidential information is not inadvertently leaked.

E.5. Replay Attacks on 0-RTT

Replayable 0-RTT data presents a number of security threats to TLS-using applications, unless those applications are specifically engineered to be safe under replay (minimally, this means idempotent, but in many cases may also require other stronger conditions, such as constant-time response). Potential attacks include:

- Duplication of actions which cause side effects (e.g., purchasing an item or transferring money) to be duplicated, thus harming the site or the user.

- Attackers can store and replay 0-RTT messages in order to re-order them with respect to other messages (e.g., moving a delete to after a create).

- Exploiting cache timing behavior to discover the content of 0-RTT messages by replaying a 0-RTT message to a different cache node and then using a separate connection to measure request latency, to see if the two requests address the same resource.

If data can be replayed a large number of times, additional attacks become possible, such as making repeated measurements of the speed of cryptographic operations. In addition, they may be able to overload rate-limiting systems. For further description of these attacks, see [Mac17].

Ultimately, servers have the responsibility to protect themselves against attacks employing 0-RTT data replication. The mechanisms described in Section 8 are intended to prevent replay at the TLS layer but do not provide complete protection against receiving multiple copies of client data. TLS 1.3 falls back to the 1-RTT handshake when the server does not have any information about the client, e.g., because it is in a different cluster which does not share state or because the ticket has been deleted as described in Section 8.1. If the application layer protocol retransmits data in this setting, then it is possible for an attacker to induce message duplication by sending the ClientHello to both the original cluster (which processes the data immediately) and another cluster which will fall back to 1-RTT and process the data upon application layer replay. The scale of this attack is limited by the client’s willingness to retry transactions and therefore only allows a limited amount of duplication, with each copy appearing as a new connection at the server.

If implemented correctly, the mechanisms described in Section 8.1 and Section 8.2 prevent a replayed ClientHello and its associated 0-RTT data from being accepted multiple times by any cluster with consistent state; for servers which limit the use of 0-RTT to one cluster for a single ticket, then a given ClientHello and its associated 0-RTT data will only be accepted once. However, if state is not completely consistent, then an attacker might be able to have multiple copies of the data be accepted during the replication window. Because clients do not know the exact details of server behavior, they MUST NOT send messages in early data which are not safe to have replayed and which they would not be willing to retry across multiple 1-RTT connections.

Application protocols MUST NOT use 0-RTT data without a profile that defines its use. That profile needs to identify which messages or interactions are safe to use with 0-RTT and how to handle the situation when the server rejects 0-RTT and falls back to 1-RTT.

In addition, to avoid accidental misuse, TLS implementations MUST NOT enable 0-RTT (either sending or accepting) unless specifically requested by the application and MUST NOT automatically resend 0-RTT data if it is rejected by the server unless instructed by the application. Server-side applications may wish to implement special processing for 0-RTT data for some kinds of application traffic (e.g., abort the connection, request that data be resent at the application layer, or delay processing until the handshake completes). In order to allow applications to implement this kind of processing, TLS implementations MUST provide a way for the application to determine if the handshake has completed.

E.5.1. Replay and Exporters

Replays of the ClientHello produce the same early exporter, thus requiring additional care by applications which use these exporters. In particular, if these exporters are used as an authentication channel binding (e.g., by signing the output of the exporter) an attacker who compromises the PSK can transplant authenticators between connections without compromising the authentication key.

In addition, the early exporter SHOULD NOT be used to generate server-to-client encryption keys because that would entail the reuse of those keys. This parallels the use of the early application traffic keys only in the client-to-server direction.

E.6. PSK Identity Exposure

Because implementations respond to an invalid PSK binder by aborting the handshake, it may be possible for an attacker to verify whether a given PSK identity is valid. Specifically, if a server accepts both external PSK and certificate-based handshakes, a valid PSK identity will result in a failed handshake, whereas an invalid identity will just be skipped and result in a successful certificate handshake. Servers which solely support PSK handshakes may be able to resist this form of attack by treating the cases where there is no valid PSK identity and where there is an identity but it has an invalid binder identically.

E.7. Attacks on Static RSA

Although TLS 1.3 does not use RSA key transport and so is not directly susceptible to Bleichenbacher-type attacks, if TLS 1.3 servers also support static RSA in the context of previous versions of TLS, then it may be possible to impersonate the server for TLS 1.3 connections [JSS15]. TLS 1.3 implementations can prevent this attack by disabling support for static RSA across all versions of TLS. In principle, implementations might also be able to separate certificates with different keyUsage bits for static RSA decryption and RSA signature, but this technique relies on clients refusing to accept signatures using keys in certificates that do not have the digitalSignature bit set, and many clients do not enforce this restriction.

Appendix F. Working Group Information

The discussion list for the IETF TLS working group is located at the e-mail address tls@ietf.org [1]. Information on the group and information on how to subscribe to the list is at https://www.ietf.org/mailman/listinfo/tls

Archives of the list can be found at: https://www.ietf.org/mail-archive/web/tls/current/index.html

Appendix G. Contributors

- Martin Abadi
  University of California, Santa Cruz
  abadi@cs.ucsc.edu

- Christopher Allen (co-editor of TLS 1.0)
  Alacrity Ventures
  ChristopherA@AlacrityManagement.com

- Richard Barnes
  Cisco
  rlb@ipv.sx

- Steven M. Bellovin
  Columbia University
  smb@cs.columbia.edu

- David Benjamin
  Google
  davidben@google.com

- Benjamin Beurdouche
  INRIA & Microsoft Research
  benjamin.beurdouche@ens.fr

- Karthikeyan Bhargavan (co-author of [RFC7627])
  INRIA
  karthikeyan.bhargavan@inria.fr

- Simon Blake-Wilson (co-author of [RFC4492])
  BCI
  sblakewilson@bcisse.com

- Nelson Bolyard (co-author of [RFC4492])
  Sun Microsystems, Inc.
  nelson@bolyard.com

- Ran Canetti
  IBM
  canetti@watson.ibm.com

- Matt Caswell
  OpenSSL
  matt@openssl.org

- Stephen Checkoway
  University of Illinois at Chicago
  sfc@uic.edu

- Pete Chown
  Skygate Technology Ltd
  pc@skygate.co.uk

- Katriel Cohn-Gordon
  University of Oxford
  me@katriel.co.uk

- Cas Cremers
  University of Oxford
  cas.cremers@cs.ox.ac.uk

- Antoine Delignat-Lavaud (co-author of [RFC7627])
  INRIA
  antdl@microsoft.com

- Tim Dierks (co-editor of TLS 1.0, 1.1, and 1.2)
  Independent
  tim@dierks.org

- Roelof DuToit
  Symantec Corporation
  roelof_dutoit@symantec.com

- Taher Elgamal
  Securify
  taher@securify.com

- Pasi Eronen
  Nokia
  pasi.eronen@nokia.com

- Cedric Fournet
  Microsoft
  fournet@microsoft.com

- Anil Gangolli
  anil@busybuddha.org

- David M. Garrett
  dave@nulldereference.com

- Illya Gerasymchuk
  Independent
  illya@iluxonchik.me

- Alessandro Ghedini
  Cloudflare Inc.
  alessandro@cloudflare.com

- Daniel Kahn Gillmor
  ACLU
  dkg@fifthhorseman.net

- Matthew Green
  Johns Hopkins University

mgreen@cs.jhu.edu

- Jens Guballa
  ETAS
  jens.guballa@etas.com

- Felix Guenther
  TU Darmstadt
  mail@felixguenther.info

- Vipul Gupta (co-author of [RFC4492])
  Sun Microsystems Laboratories
  vipul.gupta@sun.com

- Chris Hawk (co-author of [RFC4492])
  Corriente Networks LLC
  chris@corriente.net

- Kipp Hickman

- Alfred Hoenes

- David Hopwood
  Independent Consultant
  david.hopwood@blueyonder.co.uk

- Marko Horvat
  MPI-SWS
  mhorvat@mpi-sws.org

- Jonathan Hoyland
  Royal Holloway, University of London jonathan.hoyland@gmail.com

- Subodh Iyengar
  Facebook
  subodh@fb.com

- Benjamin Kaduk
  Akamai
  kaduk@mit.edu

- Hubert Kario
  Red Hat Inc.
  hkario@redhat.com

- Phil Karlton (co-author of SSL 3.0)

- Leon Klingele

Independent
mail@leonklingele.de

- Paul Kocher (co-author of SSL 3.0)
  Cryptography Research
  paul@cryptography.com

- Hugo Krawczyk
  IBM
  hugokraw@us.ibm.com

- Adam Langley (co-author of [RFC7627])
  Google
  agl@google.com

- Olivier Levillain
  ANSSI
  olivier.levillain@ssi.gouv.fr

- Xiaoyin Liu
  University of North Carolina at Chapel Hill
  xiaoyin.l@outlook.com

- Ilari Liusvaara
  Independent
  ilariliusvaara@welho.com

- Atul Luykx
  K.U. Leuven
  atul.luykx@kuleuven.be

- Colm MacCarthaigh
  Amazon Web Services
  colm@allcosts.net

- Carl Mehner
  USAA
  carl.mehner@usaa.com

- Jan Mikkelsen
  Transactionware
  janm@transactionware.com

- Bodo Moeller (co-author of [RFC4492])
  Google
  bodo@acm.org

- Kyle Nekritz

Facebook
knekritz@fb.com

- Erik Nygren
  Akamai Technologies
  erik+ietf@nygren.org

- Magnus Nystrom
  Microsoft
  mnystrom@microsoft.com

- Kazuho Oku
  DeNA Co., Ltd.
  kazuhooku@gmail.com

- Kenny Paterson
  Royal Holloway, University of London
  kenny.paterson@rhul.ac.uk

- Alfredo Pironti (co-author of [RFC7627])
  INRIA
  alfredo.pironti@inria.fr

- Andrei Popov
  Microsoft
  andrei.popov@microsoft.com

- Marsh Ray (co-author of [RFC7627])
  Microsoft
  maray@microsoft.com

- Robert Relyea
  Netscape Communications
  relyea@netscape.com

- Kyle Rose
  Akamai Technologies
  krose@krose.org

- Jim Roskind
  Amazon
  jroskind@amazon.com

- Michael Sabin

- Joe Salowey
  Tableau Software
  joe@salowey.net

- Rich Salz
  Akamai
  rsalz@akamai.com

- David Schinazi
  Apple Inc.
  dschinazi@apple.com

- Sam Scott
  Royal Holloway, University of London
  me@samjs.co.uk

- Dan Simon
  Microsoft, Inc.
  dansimon@microsoft.com

- Brian Smith
  Independent
  brian@briansmith.org

- Brian Sniffen
  Akamai Technologies
  ietf@bts.evenmere.org

- Nick Sullivan
  Cloudflare Inc.
  nick@cloudflare.com

- Bjoern Tackmann
  University of California, San Diego
  btackmann@eng.ucsd.edu

- Tim Taubert
  Mozilla
  ttaubert@mozilla.com

- Martin Thomson
  Mozilla
  mt@mozilla.com

- Sean Turner
  sn3rd
  sean@sn3rd.com

- Steven Valdez
  Google
  svaldez@google.com

- Filippo Valsorda
  Cloudflare Inc.
  filippo@cloudflare.com

- Thyla van der Merwe
  Royal Holloway, University of London
  tjvdmerwe@gmail.com

- Victor Vasiliev
  Google
  vasilvv@google.com

- Tom Weinstein

- Hoeteck Wee
  Ecole Normale Superieure, Paris
  hoeteck@alum.mit.edu

- David Wong
  NCC Group
  david.wong@nccgroup.trust

- Christopher A. Wood
  Apple Inc.
  cawood@apple.com

- Tim Wright
  Vodafone
  timothy.wright@vodafone.com

- Peter Wu
  Independent
  peter@lekensteyn.nl

- Kazu Yamamoto
  Internet Initiative Japan Inc.
  kazu@iij.ad.jp

Author's Address

Eric Rescorla
RTFM, Inc.

EMail: ekr@rtfm.com

TLS Working Group   B. Lonc
Internet-Draft   Renault
Intended status: Informational   June 12, 2015
Expires: December 14, 2015

Transport Layer Security (TLS) Authentication using ITS ETSI and IEEE certificates
draft-lonc-tls-certieee1609-01.txt

Abstract

This document specifies the use of two new certificate types to authenticate TLS entities. The first type enables the use of a certificate specified by the Institute of Electrical and Electronics Engineers (IEEE) [IEEE-ITS] and the second by the European Telecommunications Standards Institute (ETSI) [ETSI-ITS].

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

At present, TLS protocol uses X509 [RFC5246] and OpenPGP digital certificates [RFC6091] in order to authenticate servers and clients. This document describes the use of certificates specified either by the Institute of Electrical and Electronics Engineers (IEEE) [IEEE-ITS] or the European Telecommunications Standards Institute (ETSI) [ETSI-ITS]. These standards were defined in order to secure communications in vehicular environments. Existing certificates, such as X509 and OpenPGPG, are designed for Internet use, particularly for flexibility and extensibility, and are not optimized for bandwidth and processing time to support delay-sensitive applications. This is why size-optimized certificates that meet the ITS requirements were designed and standardized.

In addition, the purpose of these certificates is to provide privacy relying on geographical and/or temporal validity criteria, and minimizing the exchange of private data.

Two new values referring the previously mentioned certificated are added to the "cert_type" extension defined in [RFC6091].

2. Requirements Terminology

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].

3. Extension Overview

In order to negotiate the support of IEEE or ETSI certificate-based authentication, clients MAY include an extension of type "cert_type" in the extended client hello. The "extension_data" field of this extension SHALL contain a list of supported certificate types proposed by the client, where:

```
enum {
    X.509(0), OpenPGP(1), RawPublicKey(2),
    IEEE(3), ETSI(4), (255)
}CertificateType;
```

In case where the TLS server accepts the described extension, it selects one of the certificate types in the extension described here. The same extension type and structure will be used for the server’s response to the extension described here. Note that a server MAY send no certificate type if it either does not support it or wishes to authenticate the client using other authentication methods. The client MAY at its discretion either continue the handshake, or respond with a fatal message alert.

The end-entity certificate’s public key has to be compatible with one of the certificate types listed in extension described here.

Servers aware of the extension described here but not wishing to use it, SHOULD gracefully revert to a classical TLS handshake or decide not to proceed with the negotiation.

4. Security Considerations

This section provides an overview of the basic security considerations which need to be taken into account before implementing the necessary security mechanisms. The security considerations described throughout [RFC5246] apply here as well.

For security considerations in a vehicular environment, the minimal use of any TLS extensions is recommended such as :

- o The "cert_type" [IANA value 9] extension who’s purpose was previously described in Section 3.

- The "elliptic_curves" [IANA value 10] extension which indicates the set of elliptic curves supported by the client.

- The "SessionTicket" [IANA value 35] extension for session resumption.

In addition, servers SHOULD not support renegotiation [RFC5746] which presented Man-In-The-Middle (MITM) type attacks over the past years.

The ETSI and IEEE Standards propose the use of secp256r1 (aka NIST P-256) recommended by the NIST FIPS 186-4 standard [FIPS186].

Elliptic curve algorithms require significantly shorter public keys to achieve the same security strength. ECC is the digital signature algorithm of choice in the IEEE 1609.2 standard that specifies security services and procedures designed for vehicle communications. The ECDSA is specified in American National Standard (ANS) X9.62 . NIST approved the use of ECDSA and specified additional requirements in the FIPS Publication 186-4.

ECDSA also produces smaller signatures than RSA. The smaller key sizes and signature sizes of ECDSA mean lower message overheads when transporting ECDSA public keys over wireless networks compared with transporting RSA or DSA public keys. This is important in a large vehicle network where vehicles may often have to exchange their public keys over bandwidth - limited wireless channels. The smaller ECDSA key lengths can also translate into savings on computing power, storage and memory space, and energy required to achieve the same security strength [KARGL] [SCHUTZE] [PETIT] [ICSI]. This makes ECDSA attractive for resource - constrained mobile devices, such as vehicle on-board communication units.

The Standard defines ECIES as the encryption algorithm. Seen that this RFC aims to client authentication, the use of this algorithm can be optional for future use but not required.

AES-CCM provides both authentication and confidentiality (encryption and decryption) and uses as its only primitive the AES encrypt block cipher operation. This makes it amenable to compact implementations, which are advantageous in constrained envrionments. Adoption outside of constrained environments is necessary to enable interoperability, such as that between web clients and embedded servers, or between embedded clients and web servers.

5. IANA Considerations

Existing IANA references have not been updated yet to point to this document.

IANA is asked to register two new values in the "TLS Certificate Types" registry of Transport Layer Security (TLS) Extensions [TLS-Certificate-Types-Registry], as follows:

- Value: 3 Description: IEEE Reference: [THIS RFC]

- Value: 4 Description: ETSI Reference: [THIS RFC]

6. Cipher Suites

The table below defines ECC cipher suites that should be used [RFC7251]:

```
CipherSuite TLS_ECDHE_ECDSA_WITH_AES_128_CCM = {0xC0,0xAC}
CipherSuite TLS_ECDHE_ECDSA_WITH_AES_256_CCM = {0xC0,0xAD}
CipherSuite TLS_ECDHE_ECDSA_WITH_AES_128_CCM_8 = {0xC0,0xAE}
CipherSuite TLS_ECDHE_ECDSA_WITH_AES_256_CCM_8 = {0xC0,0xAF}
```

Figure 1: TLS ECC cipher suites

Server implementations SHOULD support all of the previous cipher suites, and client implementations SHOULD support at least one of them. Note that the versions "*_CCM_8" of cipher suites use a 64 bits tag rather than a 128 bits tag. Such cipher suites MAY be preferred in ITS networks to gain in bandwidth and message size but at the cost of a loss in integrity.

7. Message Flow

The "cert_type" message MUST be sent as the first handshake message as illustrated in Figure 1 below.

```
        Client                                               Server

        ClientHello
        /*    with
        certificate type */          -------->
                                                        ServerHello
                                                   /*    with
                                                   certificate type */
                                                         Certificate*
                                                   ServerKeyExchange*
                                                  CertificateRequest*
                                     <--------        ServerHelloDone

        Certificate
        ClientKeyExchange
        CertificateVerify*

        [ChangeCipherSpec]
        Finished                     -------->
                                                   [ChangeCipherSpec]
                                     <--------               Finished
        Application Data             <------->       Application Data
```

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

Figure 2: Message Flow with certificate type extension

7.1. Client Hello

In order to indicate the support of IEEE or ETSI certificates, clients MUST include an extension of type "cert_type" to the extended client hello message. The hello extension mechanism is described in Section 7.4.1.4 of TLS 1.2 [RFC5246].

The extension 'cert_type' sent in the client hello MAY carry a list of supported certificate types, sorted by client preference. It is a list in the case where the client supports multiple certificate types.

In a vehicular environment, privacy is important. In order to preserve anonymity, a client MUST include IEEE or ETSI certificate types in the "cert_type" extension prior to other supported certificates.

A TLS client that proposes ECC algorithms in its ClientHello message SHOULD include "elliptic_curves" extension [RFC4492].

Clients respond along with their certificates by sending a "Certificate" message immediately followed by the "ClientKeyExchange" message. The premaster secret is generated according to the cipher algorithm selected by the server in the ServerHello.cipher_suite.

7.2. Server Hello

If the server receives a client hello that contains the "cert_type" extension and chooses a cipher suite that requires a certificate, then two outcomes are possible. The server MUST either, select a certificate type from the certificate_types field in the extended client hello and must take into account the client list priority, or terminate the session with a fatal alert of type "unsupported_certificate".

The certificate type selected by the server is encoded in a CertificateTypeExtension structure, which is included in the extended server hello message using an extension of type "cert_type".

Servers implementing ECC cipher suites MUST support "elliptic_curves" extension, and when a client uses this extension, servers MUST NOT negotiate the use of an ECC cipher suite unless they can complete the handshake while respecting the choice of curves and compression techniques specified by the client [RFC4492].

7.3. Client Authentication

Client authentication is done upon specific request of the server. This procedure SHALL be done as described in section 7.4.4 of [RFC5246].

The figure below depicts the format of the CertificateRequest message.

```
enum {
    rsa_sign(1), dss_sign(2), rsa_fixed_dh(3),
    dss_fixed_dh(4), rsa_ephemeral_dh_RESERVED(5),
    dss_ephemeral_dh_RESERVED(6), fortezza_dms_RESERVED(20),
    ECDSA_sign(64), (255)
} ClientCertificateType;

opaque DistinguishedName<1..2^16-1>;

struct {
    ClientCertificateType certificate_types<1..2^8-1>;
    SignatureAndHashAlgorithm
        supported_signature_algorithms<2^16-1>;
    DistinguishedName certificate_authorities<0..2^16-1>;
} CertificateRequest;
```

Figure 3: Structure of the CertificateRequest message

The CertificateRequest SHALL be filled as follow:

ClientCertificateType   ECDSA_sign(64)

SignatureAndHashAlgorithm {0x04,0x03} (ECDSA-SHA256)

DistinguishedName   It MAY be used by the server to specify a list of certificate authorities it trusts (i.e. AA/PCA or EA/LTCA). If possible, the client SHOULD then reply with a certificate signed by one of the certificate authorities trusted by the server in order to avoid sending the certificate chain. A certificate authority is identified by its HashedId8 as defined in section 4.2.12 of [ETSI-ITS]. That is, DistinguishedName is a list of HashedId8. If not used this field MUST be empty.

8. Certificate Verification

8.1. IEEE 1609.2 certificates

Verification of an IEEE 1609.2 certificate or certificate chain is described in section 5.5.2 of [IEEE-ITS].

8.2. ETSI TS 103 097 certificates

The format of an ETSI TS 103 097 certificate is depicted in the figure below.

```
+------+-------+--------+----------+-------+-----+----------+------+
|      |       |        | verifi-  |assu-  |its  | validity |      |
| ver- |signer |subject | cation/  |rance  |_aid | _restri- |sign- |
| sion |_info  |_info   | encryp-  |_level |_ssp | ctions   |ature |
|      |       |        | tion key |       |     |          |      |
+------+-------+--------+----------+-------+-----+----------+------+
```

Figure 4: ETSI TS 103 097 certificate format

The verification process of an ETSI TS 103 097 certificate SHALL follow these steps:

1. Verify that the certificate content is conform to one of ETSI profiles (RCA, EA, AA, EC, AT). If not, the verification has failed and the message SHALL be discarded.

2. Verify the certificate’s signer identity:

   * If the certificate digest included in "signer_info" is known, goto step 3.

   * Else:

     + If it is a root certificate digest, the verification has failed (error - untrusted RCA) and the message SHALL be discarded.

     + Else: pause the current certificate verification process and start verification of the next certificate in the chain (which SHALL be the signer’s certificate) recursively by restarting from step 1. Once verified, resume the certificate verification previously paused.

3. Verify that the certificate is not in the Certificate Revocation List (CRL). If it is, the verification has failed (error - certificate in CRL) and the message SHALL be discarded.

4. Verify the signature of the certificate (see [RFC4492] for details).

5. Verify "subject_info": "subject_name" SHALL be a 32 bytes hash of the server URL. Note that this step is only done by clients that

are verifying a server’s certificate. In the opposite case this step SHALL be ignored.

6. Verify "validity_restrictions": only the validity of time is checked, the validity of space (i.e. geographical region) is ignored.

7. Verify the "its_aid_ssp": ITS-AID included in the certificate SHALL be consistent with those included in the signer’s certificate (heritage).

9. IEEE - ETSI comparison

The ETSI and IEEE 1609.2 represent the active standardization groups in Europe and U.S those dealing with the security of vehicular communications. Although defined for the same purpose, the different security requirements have led to the definition of different certificate formats.

9.1. Certificate Encoding

As described in the IEEE 1609.2 and ETSI standards, the internal representation of the certificate structure is encoded into a flat octet string in network byte order (i.e. big-endian).

IEEE 1609.2 is developing for future an ASN.1 version of the standard using X.696 (OER) [X696].

10. References

10.1. Normative References

[ETSI-ITS]
ETSI, , "ETSI TS 103 097 v1.1.1 (2013-04): Intelligent Transport Systems (ITS); Security; Security header and certificate formats", April 2013.

[IEEE-ITS]
IEEE 1609.2, , "IEEE Standard for Wireless Access in Vehicular Environments - Security Services for Applications and Management Messages", 2013.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", March 1997.

[RFC4492] Blake-Wilson, S., Bolyard, N., Gupta, V., Hawk, C., and B. Moeller, "Elliptic Curve Cryptography (ECC) Cipher Suites for Transport Layer Security (TLS)", May 2006.

[RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", August 2008.

[RFC5746] Rescorla, E., Ray, M., Dispensa, S., and N. Oskov, "Transport Layer Security (TLS) Renegotiation Indication Extension"", February 2010.

[RFC6091] Mavrogiannopoulos, N. and D. Gillmor, "Using OpenPGP Keys for Transport Layer Security (TLS) Authentication", February 2011.

[RFC7251] McGrew, D., Bailey, D., Campagna, M., and R. Dugal, "AES-CCM Elliptic Curve Cryptography (ECC) Cipher Suites for TLS", June 2014.

10.2. Informative References

[FIPS186] FIPS 186-4, , "Digital Signature Standard", July 2013.

[KARGL] Kargl, F., Papadimitratos, P., Buttyan, L., Muter, M., Schoch, E., Wiedersheim, B., Thong, T., Calandriello, G., Held, A., Kung, A., and J. Hubaux, "Secure Vehicular Communications: Implementation, Performance, and Research Challenges", November 2008.

[SCHUTZE] Schutze, T., "Automotive security: Cryptography for Car2X communication", March 2011.

[PETIT] Petit, J., "Analysis of ECDSA authentication processing in VANETs", December 2009.

[ICSI] ICST project, , "Analysis of timeliness of communication for IEEE 1609.2", 2013.

[X696] ITU-T X.696, , "Information Technology - ASN.1 encoding rules: Specification of Octet Encoding Rules (OER)", august 2014.

Appendix A. ETSI Encoding Example

The hex sequence shown in Figure 5 presents an encoded secured message with signed payload as a generic encoded octet string.

```
        00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15
      +---+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--
   01 | 02 80 ba 80 02 02 01 53 88 de c6 40 c6 e1 9e 01
   02 | 00 52 00 00 04 d4 81 34 8a cd d1 d9 9c 1f fb a4
   03 | c7 0e 6d 2a 5d 13 ca b0 a1 e6 cf 63 22 9f 69 79
   04 | b4 53 c0 15 c7 da 3a 12 7c 8f 39 44 59 b1 2f 94
   05 | d4 cb 9a 12 ce e1 1d 87 40 8d 91 ac 95 6c 90 c8
   06 | b3 b2 9f 4c 22 02 e0 21 0b 24 03 01 00 00 25 04
   07 | 01 00 00 00 0b 01 15 04 39 83 15 4c bc 02 03 00
   08 | 00 00 71 ff 9a 0d 80 16 ca cb cd d8 1c d1 4f 81
   09 | 94 3c dd c7 74 51 1e 2b f7 15 7b 33 e5 4f 7b 6b
   10 | 6e 5b 5d 07 94 70 be 40 a6 46 e0 55 9c 19 89 28
   11 | b5 b8 ed cf bd c2 29 70 53 95 1d bc 51 cb d6 a3
   12 | e1 d0 00 00 01 41 ae 0f 26 64 c0 05 24 01 55 20
   13 | 50 02 80 00 31 01 00 14 00 30 14 4a d9 f8 7e 59
   14 | 9e 09 2b 00 00 00 00 00 00 00 00 80 00 00 00 00
   15 | 00 00 00 07 d1 00 00 01 02 00 00 00 02 09 2b 40
   16 | 56 b4 9d 20 0d 69 3a 40 1f ff ff fc 22 30 d4 1e
   17 | 40 00 0f c0 00 7e 02 76 ea 87 33 a9 d7 4f ff d0
   18 | 84 14 00 00 43 01 00 00 61 6d 42 37 dd 2c ea b7
   19 | 27 31 c2 3b cb 5d 61 8f 88 17 df 0d a8 7b d2 b8
   20 | d3 54 8f 71 09 8a f1 88 d2 43 04 a8 61 6a 95 bf
   21 | 5e 07 45 a1 06 e9 33 9f 9e 69 ba b3 3c bc 68 28
   22 | 93 5a 66 ea 11 a0 37 69
```

Figure 5: Example of encoded ETSI secured message with signed payload

In the parsed data structure, the contents are presented in the form:

```
struct SecuredMessage {
  uint8 protocol_version: 2
  HeaderField<186> header_fields {
    struct HeaderField {
      HeaderFieldType type: signer_info (128)
      struct SignerInfo signer {
        SignerInfoType type: certificate (2)
        struct Certificate certificate {
          uint8 version: 2
          struct SignerInfo signer_info {
            SignerInfoType type: certificate_digest_with_sha256 (1)
            HashedId8 digest: 5388DEC640C6E19E
          }
          struct SubjectInfo subject_info {
```

```
            SubjectType subject_type: authorization_ticket (1)
            opaque<0> subject_name:
          }
          SubjectAttribute<82> subject_attributes {
            struct SubjectAttribute {
              SubjectAttributeType type: verification_key (0)
              struct PublicKey key {
                PublicKeyAlgorithm algorithm: ecdsa_nistp256_with_
                                              sha256 (0)
                struct EccPoint public_key {
                  EccPointType type: uncompressed (4)
                  opaque[32] x: D481348ACDD1D99C1FFBA4C70E6D2A5D
                                13CAB0A1E6CF63229F6979B453C015C7
                  opaque[32] y: DA3A127C8F394459B12F94D4CB9A12CE
                                E11D87408D91AC956C90C8B3B29F4C22
                }
              }
            }
            struct SubjectAttribute {
              SubjectAttributeType type: assurance_level (2)
              SubjectAssurance assurance_level: assurance level = 7,
                                                confidence = 0
                                                (bitmask = 11100000)
            }
            struct SubjectAttribute {
              SubjectAttributeType type: its_aid_ssp_list (33)
              ItsAidSsp<11> its_aid_ssp_list {
                struct ItsAidSsp {
                  IntX its_aid: 36
                  opaque<3> service_specific_permissions: 010000
                }
                struct ItsAidSsp {
                  IntX its_aid: 37
                  opaque<4> service_specific_permissions: 01000000
                }
              }
            }
          }
          ValidityRestriction<11> validity_restrictions {
            struct ValidityRestriction {
              ValidityRestrictionType type: time_start_and_end (1)
              Time32 start_validity: 2015-03-05 00:00:00 UTC
              Time32 end_validity: 2015-04-28 23:59:59 UTC
            }
            struct ValidityRestriction {
              ValidityRestrictionType type: region (3)
              struct GeographicRegion region {
                RegionType region_type: none (0)
```

```
              }
            }
          }
          struct Signature {
            PublicKeyAlgorithm algorithm: ecdsa_nistp256_with_sha256 (0)
            struct EcdsaSignature ecdsa_signature {
              struct EccPoint R {
                EccPointType type: x_coordinate_only (0)
                opaque[32] x: 71FF9A0D8016CACBCDD81CD14F81943C
                              DDC774511E2BF7157B33E54F7B6B6E5B
              }
              opaque[32] s: 5D079470BE40A646E0559C198928B5B8
                            EDCFBDC2297053951DBC51CBD6A3E1D0
            }
          }
        }
      }
    }
    struct HeaderField {
      HeaderFieldType type: generation_time (0)
      Time64 generation_time: 2015-03-17 15:26:48.000 UTC
    }
    struct HeaderField {
      HeaderFieldType type: its_aid (5)
      IntX its_aid: 36
    }
  }
  struct Payload payload_field {
    PayloadType type: signed (1)
    opaque<85> data: 20500280003101001400301440AD9F87E
                     599E092B0000000000000000080000000
                     0000000007D100000102000000002092B
                     4056B49D200D693A401FFFFFFC2230D4
                     1E40000FC0007E0276EA8733A9D74FFF
                     D084140000
  }
  TrailerField<67> trailer_fields {
    struct TrailerField {
      TrailerFieldType type: signature (1)
      struct Signature signature {
        PublicKeyAlgorithm algorithm: ecdsa_nistp256_with_sha256 (0)
        struct EcdsaSignature ecdsa_signature {
          struct EccPoint R {
            EccPointType type: x_coordinate_only (0)
            opaque[32] x: 616D4237DD2CEAB72731C23BCB5D618F
                          8817DF0DA87BD2B8D3548F71098AF188
          }
          opaque[32] s: D24304A8616A95BF5E0745A106E9339F
```

```
                        9E69BAB33CBC6828935A66EA11A03769
        }
      }
    }
  }
}
```

Figure 6: Example of parsed ETSI secured message with signed payload

Appendix B. IEEE Encoding Example

The hex sequence shown in Figure 7 presents an encoded signed data structure as a flat encoded octet string.

```
          00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15
        +---+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--
     01 | 02 01 03 02 02 04 f3 db 4f 6f ca b6 49 65 01 09
     02 | 63 65 72 74 4e 61 6d 65 31 01 05 e0 00 00 01 00
     03 | 04 00 00 00 00 00 00 00 01 00 02 d4 a8 61 1d ce
     04 | d8 8c a7 a2 e9 6a 8d 7e 49 0f 3c 9a 46 27 c0 72
     05 | 26 ed 67 8d 04 74 41 02 00 03 9c b6 6f 87 4a 40
     06 | 7c 21 83 40 22 db 6d 0a 80 d0 14 cb df 24 fc a0
     07 | 83 f8 e2 00 81 b0 7c 14 b8 e7 02 19 90 d0 57 4b
     08 | 14 d2 80 29 1f c4 e6 a6 73 12 68 74 96 77 c2 52
     09 | 34 ae bb e4 29 da 16 60 61 19 74 c6 b3 53 98 0e
     10 | 70 e3 3d 4f b9 03 99 76 05 44 e9 74 70 d9 92 bb
     11 | 3c 37 92 c3 51 d4 7d 8e ea b1 03 0a e0 00 00 01
     12 | 0c 73 6f 6d 65 20 63 6f 6e 74 65 6e 74 00 00 e7
     13 | 2a dc 3e dc 09 00 00 00 00 00 00 00 00 00 00 00
     14 | 02 ca bf a2 0d 82 ae 3e 25 a3 8c 9c dd 2e cf 94
     15 | 9f cc 7c 7f d9 d8 83 89 f5 08 f7 aa bb 5b ef 21
     16 | bd 7a 2e 79 6c c7 de 01 af b1 93 35 5b e2 f5 88
     17 | 19 76 70 e4 ae 09 cf 3b ee
```

Figure 7: Example of encoded IEEE 1609.2 signed data structure

In the parsed data structure, the contents are presented in the form:

```
protocol_version (0, 1): 02
type (1, 1): 01 (signed)
signed_data (2, 263):
  signer (2, 169):
    type (2, 1): 03 (certificate)
    certificates (3, 168):
      version_and_type (3, 1): 02 (explicit)
      unsigned_certificate (4, 102):
        holder_type (4, 1): 02 (identified localized)
```

```
              cf (5, 1): 04 (encryption_key)
              signer_id (6, 8): f3 db 4f 6f ca b6 49 65
              signature_alg (14, 1): 01 (ECDSA NIST P256)
              scope (15, 18):
                id_scope (15, 18):
                  name_len (15, 1): 09
                  name (16, 9): 63 65 72 74 4e 61 6d 65 31
                  permissions (25, 7):
                    type (25, 1): 01 (specified)
                    permissions_list_len (26, 1): 05
                    permissions_list (27, 5):
                      psid (27, 4): e0 00 00 01
                      service_specific_permissions_len (31, 1): 00
                  region (32, 1):
                    region_type (32, 1): 04 (none)
              expiration (33, 4): 00 00 00 00 (00:00:34 01 Jan 2004 UTC)
              crl_series (37, 4): 00 00 00 01
              verification_key (41, 30):
                algorithm (41, 1): 00 (ECDSA NIST P224)
                public_key (42, 29):
                  type (42, 1): 02 (compressed, lsb of y is 0)
                  x (43, 28):
                    d4 a8 61 1d ce d8 8c a7 a2 e9 6a 8d 7e 49 0f 3c
                    9a 46 27 c0 72 26 ed 67 8d 04 74 41
              encryption_key (71, 35):
                algorithm (71, 1): 02 (ECIES NIST P256)
                supported_symm_alg (72, 1): 00 (AES 128 CCM)
                public_key (73, 33):
                  type (73, 1): 03 (compressed, lsb of y is 1)
                  x (74, 32):
                    9c b6 6f 87 4a 40 7c 21 83 40 22 db 6d 0a 80 d0
                    14 cb df 24 fc a0 83 f8 e2 00 81 b0 7c 14 b8 e7
            signature (106, 65):
              ecdsa_signature (106, 65):
                R (106, 33):
                  type (106, 1): 02 (compressed, lsb of y is 0)
                  x (107, 32):
                    19 90 d0 57 4b 14 d2 80 29 1f c4 e6 a6 73 12 68
                    74 96 77 c2 52 34 ae bb e4 29 da 16 60 61 19 74
                  s (139, 32):
                    c6 b3 53 98 0e 70 e3 3d 4f b9 03 99 76 05 44 e9
                    74 70 d9 92 bb 3c 37 92 c3 51 d4 7d 8e ea b1 03
      unsigned_data (171, 37):
        tf (171, 1): 0a (use_generation_time, use_location)
        psid (172, 4): e0 00 00 01
        data_len (176, 1): 0c
        data (177, 12): 73 6f 6d 65 20 63 6f 6e 74 65 6e 74
        generation_time (189, 9):
```

```
          time (189, 8): 00 00 e7 2a dc 3e dc 09
               (19:08:23 20 Jan 2012 UTC)
          log_std_dev (197, 1): 00 (1.134666 ns or less)
        generation_location (198, 10):
          latitude (198, 4): 00 00 00 00
          longitude (202, 4): 00 00 00 00
          elevation (206, 2): 00 00
      signature (208, 57):
        ecdsa_signature (208, 57):
          R (208, 29):
            type (208, 1): 02 (compressed, lsb of y is 0)
            x (209, 28):
              ca bf a2 0d 82 ae 3e 25 a3 8c 9c dd 2e cf 94 9f
              cc 7c 7f d9 d8 83 89 f5 08 f7 aa bb
            s (237, 28):
              5b ef 21 bd 7a 2e 79 6c c7 de 01 af b1 93 35 5b
              e2 f5 88 19 76 70 e4 ae 09 cf 3b ee
```

Figure 8: Example of parsed IEEE 1609.2 signed data structure

Appendix C. Co-authors’ Addresses

Houda Labiod
Telecom Paristech
46 rue Barrault
75634 Paris cedex 13
France
Email: houda.labiod@telecom-paristech.fr

Francois Lonc
Telecom Paristech
46 rue Barrault
75634 Paris cedex 13
France
Email: francois.lonc@telecom-paristech.fr

Ahmed Serhrouchni
Telecom Paristech
46 rue Barrault
75634 Paris cedex 13
France
Email: ahmed.serhrouchni@telecom-paristech.fr

Arnaud Kaiser
IRT SystemX
8 avenue de la Vauve
91120 Palaiseau

France
Email: arnaud.kaiser@irt-systemx.fr

Author’s Address

Brigitte Lonc
Renault
1 avenue du Golf
78288 Guyancourt
France

EMail: brigitte.lonc@renault.com

Network Working Group J. Mattsson
Internet-Draft D. Migault
Intended status: Standards Track Ericsson
Expires: October 20, 2016 April 18, 2016

ECDHE_PSK with AES-GCM and AES-CCM Cipher Suites for Transport Layer Security (TLS)
draft-mattsson-tls-ecdhe-psk-aead-05

Abstract

This document defines several new cipher suites for the Transport Layer Security (TLS) protocol. The cipher suites are all based on the Ephemeral Elliptic Curve Diffie-Hellman with Pre-Shared Key (ECDHE_PSK) key exchange together with the Authenticated Encryption with Associated Data (AEAD) algorithms AES-GCM and AES-CCM. PSK provides light and efficient authentication, ECDHE provides perfect forward secrecy, and AES-GCM and AES-CCM provides encryption and integrity protection.

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

This document defines new cipher suites that provide Pre-Shared Key (PSK) authentication, Perfect Forward Secrecy (PFS), and Authenticated Encryption with Associated Data (AEAD). The cipher suites are defined for version 1.2 or later of the the Transport Layer Security (TLS) [RFC5246] protocol, as well as version 1.2 or later of the Datagram Transport Layer Security (DTLS) protocol [RFC6347].

Pre-Shared Key (PSK) Authentication is widely used in many scenarios. One deployment is 3GPP networks where pre-shared keys are used to authenticate both subscriber and network. Another deployment is Internet of Things where PSK authentication is often preferred for performance and energy efficiency reasons. In both scenarios the endpoints are owned/controlled by a party that provisions the pre-shared keys and makes sure that they provide a high level of entropy.

Perfect Forward Secrecy (PFS) is a strongly recommended feature in security protocol design and can be accomplished by using an ephemeral Diffie-Hellman key exchange method. Ephemeral Elliptic Curve Diffie-Hellman (ECDHE) provides PFS with excellent performance and small key sizes. ECDHE is mandatory to implement in both HTTP/2 [RFC7540] and CoAP [RFC7252].

AEAD algorithms that combine encryption and integrity protection are strongly recommended [RFC7525] and non-AEAD algorithms are forbidden to use in TLS 1.3 [I-D.ietf-tls-tls13]. The AEAD algorithms considered in this document are AES-GCM and AES-CCM. The use of AES-

GCM in TLS is defined in [RFC5288] and the use of AES-CCM is defined in [RFC6655].

[RFC4279] defines Pre-Shared Key (PSK) cipher suites for TLS but does not consider Elliptic Curve Cryptography. [RFC4492] introduces Elliptic Curve Cryptography for TLS but does not consider PSK authentication. [RFC5487] describes the use of AES-GCM in combination with PSK authentication, but does not consider ECDHE. [RFC5489] describes the use of PSK in combination with ECDHE but does not consider AES-GCM or AES-CCM.

2. ECDHE_PSK with AES-GCM and AES-CCM Cipher Suites

The cipher suites defined in this document are based on the AES-GCM and AES-CCM Authenticated Encryption with Associated Data (AEAD) algorithms AEAD_AES_128_GCM, AEAD_AES_256_GCM, AEAD_AES_128_CCM, and AEAD_AES_256_CCM defined in [RFC5116], AEAD_AES_128_CCM_8 and AEAD_AES_256_CCM_8 defined in [RFC6655]. The following cipher suites are defined:

```
TLS_ECDHE_PSK_WITH_AES_128_GCM_SHA256   = {0xTBD,0xTBD};
TLS_ECDHE_PSK_WITH_AES_256_GCM_SHA384   = {0xTBD,0xTBD};
TLS_ECDHE_PSK_WITH_AES_128_CCM_8_SHA256 = {0xTBD,0xTBD};
TLS_ECDHE_PSK_WITH_AES_256_CCM_8_SHA256 = {0xTBD,0xTBD};
TLS_ECDHE_PSK_WITH_AES_128_CCM_SHA256   = {0xTBD,0xTBD};
TLS_ECDHE_PSK_WITH_AES_256_CCM_SHA384   = {0xTBD,0xTBD};
```

For the AES-128 cipher suites, the TLS Pseudorandom Function (PRF) with SHA-256 as the hash function SHALL be used and Clients and Servers MUST NOT negotiate curves of less than 255 bits.

For the AES-256 cipher suites, the TLS PRF with SHA-384 as the hash function SHALL be used and Clients and Servers MUST NOT negotiate curves of less than 384 bits.

When used in TLS 1.2, the keying material is derived as described in [RFC5489] and [RFC5246] and nonces are constructed as described in [RFC5288], and [RFC6655]. When used in TLS 1.3, the keying material is derived as described in [I-D.ietf-tls-tls13], and the nonces are constructed as described in [I-D.ietf-tls-tls13].

3. Applicable TLS Versions

The cipher suites defined in this document make use of the authenticated encryption with additional data (AEAD) defined in TLS 1.2 [RFC5246] and DTLS 1.2 [RFC6347]. Earlier versions of TLS do not have support for AEAD and consequently, these cipher suites MUST NOT be negotiated in TLS versions prior to 1.2. Clients MUST NOT offer these cipher suites if they do not offer TLS 1.2 or later. Servers, which select an earlier version of TLS MUST NOT select one of these cipher suites. A client MUST treat the selection of these cipher suites in combination with a version of TLS that does not support AEAD (i.e., TLS 1.1 or earlier) as an error and generate a fatal 'illegal_parameter' TLS alert.

4. IANA Considerations

This document defines the following new cipher suites, whose values have been assigned in the TLS Cipher Suite Registry defined by [RFC5246].

```
TLS_ECDHE_PSK_WITH_AES_128_GCM_SHA256   = {0xTBD; 0xTBD} {0xD0,0x01};
TLS_ECDHE_PSK_WITH_AES_256_GCM_SHA384   = {0xTBD; 0xTBD} {0xD0,0x02};
TLS_ECDHE_PSK_WITH_AES_128_CCM_8_SHA256 = {0xTBD; 0xTBD} {0xD0,0x03};
TLS_ECDHE_PSK_WITH_AES_256_CCM_8_SHA256 = {0xTBD; 0xTBD} {0xD0,0x04};
TLS_ECDHE_PSK_WITH_AES_128_CCM_SHA256   = {0xTBD; 0xTBD} {0xD0,0x05};
TLS_ECDHE_PSK_WITH_AES_256_CCM_SHA384   = {0xTBD; 0xTBD} {0xD0,0x06};
```

The cipher suite numbers listed in the second column are numbers used for cipher suite interoperability testing and it's suggested that IANA use these values for assignment.

5. Security Considerations

The security considerations in TLS 1.2 [RFC5246], DTLS 1.2 [RFC6347], TLS 1.3 [I-D.ietf-tls-tls13], ECDHE_PSK [RFC5489], AES-GCM [RFC5288], and AES-CCM [RFC6655] apply to this document as well.

All the cipher suites defined in this document provide confidentiality, mutual authentication, and perfect forward secrecy. The AES-128 cipher suites provide 128-bit security and the AES-256 cipher suites provide at least 192-bit security. However, AES_128_CCM_8 only provides 64-bit security against message forgery and AES_256_GCM and AES_256_CCM only provide 128-bit security against message forgery.

Use of Pre-Shared Keys of limited entropy (for example, a PSK that is relatively short, or was chosen by a human and thus may contain less entropy than its length would imply) may allow an active attacker to perform a brute-force attack where the attacker attempts to connect to the server and tries different keys. Passive eavesdropping alone is not sufficient. For these reasons the Pre-Shared Keys used for authentication MUST have a security level equal or higher than the cipher suite used, i.e. at least 128-bit for the AES-128 cipher suites and at least 192-bit for the AES-256 cipher suites.

6. Acknowledgements

The authors would like to thank Ilari Liusvaara, Eric Rescorla, Dan Harkins, Russ Housley and Sean Turner for their valuable comments and feedback.

7. References

7.1. Normative References

[I-D.ietf-tls-tls13] Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", draft-ietf-tls-tls13-12 (work in progress), March 2016.

[RFC4279] Eronen, P., Ed. and H. Tschofenig, Ed., "Pre-Shared Key Ciphersuites for Transport Layer Security (TLS)", RFC 4279, DOI 10.17487/RFC4279, December 2005, <http://www.rfc-editor.org/info/rfc4279>.

[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>.

[RFC5116] McGrew, D., "An Interface and Algorithms for Authenticated Encryption", RFC 5116, DOI 10.17487/RFC5116, January 2008, <http://www.rfc-editor.org/info/rfc5116>.

[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>.

[RFC5288] Salowey, J., Choudhury, A., and D. McGrew, "AES Galois Counter Mode (GCM) Cipher Suites for TLS", RFC 5288, DOI 10.17487/RFC5288, August 2008, <http://www.rfc-editor.org/info/rfc5288>.

[RFC5489] Badra, M. and I. Hajjeh, "ECDHE_PSK Cipher Suites for Transport Layer Security (TLS)", RFC 5489, DOI 10.17487/RFC5489, March 2009, <http://www.rfc-editor.org/info/rfc5489>.

[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>.

[RFC6655] McGrew, D. and D. Bailey, "AES-CCM Cipher Suites for Transport Layer Security (TLS)", RFC 6655, DOI 10.17487/RFC6655, July 2012, <http://www.rfc-editor.org/info/rfc6655>.

7.2. Informative References

[RFC5487] Badra, M., "Pre-Shared Key Cipher Suites for TLS with SHA-256/384 and AES Galois Counter Mode", RFC 5487, DOI 10.17487/RFC5487, March 2009, <http://www.rfc-editor.org/info/rfc5487>.

[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>.

[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>.

Authors’ Addresses

John Mattsson
Ericsson AB
SE-164 80 Stockholm
Sweden

Phone: +46 76 115 35 01
Email: john.mattsson@ericsson.com

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

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

TLS
Internet-Draft
Intended status: Standards Track
Expires: April 21, 2016

M. Shore
No Mountain Software
R. Barnes
Mozilla
S. Huque
Verisign Labs
W. Toorop
NLNet Labs
October 19, 2015

A DANE Record and DNSSEC Authentication Chain Extension for TLS
draft-shore-tls-dnssec-chain-extension-02

Abstract

This draft describes a new TLS extension for transport of a DNS record set serialized with the DNSSEC signatures needed to authenticate that record set. The intent of this proposal is to allow TLS clients to perform DANE authentication of a TLS server certificate without needing to perform additional DNS record lookups. It will typically not be used for general DNSSEC validation of TLS endpoint names.

Status of This Memo

Copyright Notice

Table of Contents

1. Requirements Notation

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. Introduction

This draft describes a new TLS [RFC5246] extension for transport of a DNS record set serialized with the DNSSEC signatures [RFC4034] needed to authenticate that record set. The intent of this proposal is to allow TLS clients to perform DANE authentication [RFC6698] of a TLS server certificate without performing perform additional DNS record lookups and incurring the associated latency penalty. It also provides the ability to avoid potential problems with TLS clients being unable to look up DANE records because of an interfering or broken middlebox on the path between the endpoint and a DNS server. And lastly, it allows a TLS client to validate DANE records itself without needing access to a validating DNS resolver to which it has a secure connection. It will typically not be used for general DNSSEC validation of endpoint names, but is more appropriate for validation of DANE TLSA records.

This mechanism is useful for TLS applications that need to address the problems described above, typically web browsers or VoIP and XMPP services. It may not be relevant for many other applications. For example, SMTP MTAs are usually located in data centers, may tolerate extra DNS lookup latency, are on servers where it is easier to provision a validating resolver, or are less likely to experience traffic interference from misconfigured middleboxes. Furthermore, SMTP MTAs usually employ Opportunistic Security [RFC7435], in which the presence of the DNS TLSA records is used to determine whether to enforce an authenticated TLS connection. Hence DANE authentication of SMTP MTAs [RFC7672] will typically not use this mechanism.

The extension described here allows a TLS client to request in the client hello message that the DNS authentication chain be returned in the (extended) server hello message. If the server is configured for DANE authentication, then it performs the appropriate DNS queries, builds the authentication chain, and returns it to the client. The server will usually use a previously cached authentication chain, but it will need to rebuild it periodically as described in Section 5. The client then authenticates the chain using a pre-configured trust anchor.

This specification is based on Adam Langley’s original proposal for serializing DNSSEC authentication chains and delivering them in an X.509 certificate extension [AGL]. It modifies the approach by using wire format DNS records in the serialized data (assuming that the data will be prepared and consumed by a DNS-specific library), and by using a TLS extension to deliver the data.

3. DNSSEC Authentication Chain Extension

3.1. Protocol

A client MAY include an extension of type "dnssec_chain" in the (extended) ClientHello. The "extension_data" field of this extension MUST be empty.

Servers receiving a "dnssec_chain" extension in the client hello, and which are capable of being authenticated via DANE, SHOULD return a serialized authentication chain in the extended ServerHello message, using the format described below. If a server is unable to return a authentication chain, or does not wish to return a authentication chain, it does not include a dnssec_chain extension. As with all TLS extensions, if the server does not support this extension it will not return any authentication chain.

3.2. DNSSEC Authentication Chain Data

The "extension_data" field of the "dnssec_chain" extension MUST contain a DNSSEC Authentication Chain encoded in the following form:

```
opaque AuthenticationChain<0..2^16-1>;
```

The AuthenticationChain structure is composed of a sequence of uncompressed wire format DNS resource record sets (RRset) and corresponding signatures (RRsig) records. The record sets and signatures are presented in validation order, starting at the target DANE record, followed by the DNSKEY and DS record sets for each intervening DNS zone up to a trust anchor chosen by the server, typically the DNS root.

This sequence of native DNS wire format records enables easier generation of the data structure on the server and easier verification of the data on client by means of existing DNS library functions. However this document describes the data structure in sufficient detail that implementers if they desire can write their own code to do this.

[TODO: mention that to reduce the size of the chain, the server can deliver exactly one RRsig per RRset, namely the one used to validate the chain as it is built.]

Each RRset in the chain is composed of a sequence of wire format DNS resource records. The format of the resource record is described in RFC 1035 [RFC1035], Section 3.2.1. The resource records SHOULD be presented in the canonical form and ordering as described in RFC 4034 [RFC4034].

```
RR(i) = owner | type | class | TTL | RDATA length | RDATA
```

RRs within the RRset are ordered canonically, by treating the RDATA portion of each RR as a left-justified unsigned octet sequence in which the absence of an octet sorts before a zero octet.

The RRsig record is in DNS wire format as described in RFC 4034 [RFC4034], Section 3.1. The signature portion of the RDATA, as described in the same section, is the following:

```
signature = sign(RRSIG_RDATA | RR(1) | RR(2)... )
```

where, RRSIG_RDATA is the wire format of the RRSIG RDATA fields with the Signer's Name field in canonical form and the signature field excluded.

The first RRset in the chain MUST contain the DANE records being presented. The subsequent RRsets MUST be a sequence of DNSKEY and DS RRsets, starting with a DNSKEY RRset. Each RRset MUST authenticate the preceding RRset:

A DNSKEY RRset must include the DNSKEY RR containing the public key used to verify the previous RRset.

For a DS RRset, the set of key hashes MUST overlap with the preceding set of DNSKEY records.

In addition, a DNSKEY RRset followed by a DS RRset MUST be self-signed, in the sense that its RRSIG MUST verify under one of the keys in the DNSKEY RRSET.

The final DNSKEY RRset in the authentication chain, containing the trust anchor may be omitted. If omitted, the client MUST verify that the key tag and owner name in the final RRSIG record correspond to a trust anchor. There may however be reason to include the trust anchor RRset and signature if clients are expected to use RFC5011 compliant key rollover functions inband via the chain data. In that case, they will need to periodically inspect flags (revocation and secure entry point flags) on the trust anchor DNSKEY RRset.

For example, for an HTTPS server at www.example.com, where there are zone cuts at "com." and "example.com.", the AuthenticationChain structure would comprise the following RRsets and signatures (the data field of the records are omitted here for brevity):

```
_443._tcp.www.example.com. TLSA
RRSIG(_443._tcp.www.example.com. TLSA)
```

```
example.com. DNSKEY
RRSIG(example.com. DNSKEY)
example.com. DS
RRSIG(example.com. DS)
com. DNSKEY
RRSIG(com. DNSKEY)
com. DS
RRSIG(com. DS)
. DNSKEY
RRSIG(. DNSKEY)
```

Names that are aliased via CNAME and/or DNAME records may involve multiple branches of the DNS tree. In this case the authentication chain structure will be composed of a sequence of these multiple intersecting branches. DNAME chains should omit unsigned CNAME records that may have been synthesized in the response from a DNS resolver. Wildcard DANE records will need to include the wildcard name as well as a negative proof (i.e. NSEC or NSEC3 records) that no closer name exists.

A CNAME example:

```
_443._tcp.www.example.com.   IN   CNAME    ca.example.net.
ca.example.net.              IN   TLSA     2 0 1 ...
```

Here the authentication chain structure is composed of two consecutive chains, one for _443._tcp.www.example.com/CNAME and one for ca.example.net/TLSA. The second chain can omit the record sets at the end that overlap with the first.

TLS DNSSEC chain components:

```
_443._tcp.www.example.com. CNAME
RRSIG(_443._tcp.www.example.com. CNAME)
example.com. DNSKEY
RRSIG(example.com. DNSKEY)
example.com. DS
RRSIG(example.com. DS)
com. DNSKEY
RRSIG(com. DNSKEY)
com. DS
RRSIG(com. DS)
. DNSKEY
RRSIG(. DNSKEY)

ca.example.net. TLSA
```

```
RRSIG(ca.example.net. TLSA)
example.net. DNSKEY
RRSIG(example.net. DNSKEY)
example.net. DS
RRSIG(example.net. DS)
net. DNSKEY
RRSIG(net. DNSKEY)
net. DS
RRSIG(net. DS)
```

4. Construction of Serialized Authentication Chains

This section describes a possible procedure for the server to use to build the serialized DNSSEC chain.

When the goal is to perform DANE authentication [RFC6698] of the server's X.509 certificate, the DNS record set to be serialized is a TLSA record set corresponding to the server's domain name.

The domain name of the server MUST be that included in the TLS Server Name Indication extension [RFC6066] when present. If the Server Name Indication extension is not present, or if the server does not recognize the provided name and wishes to proceed with the handshake rather than to abort the connection, the server uses the domain name associated with the server IP address to which the connection has been established.

The TLSA record to be queried is constructed by prepending the _port and _transport labels to the domain name as described in [RFC6698], where "port" is the port number associated with the TLS server. The transport is "tcp" for TLS servers, and "udp" for DTLS servers. The port number label is the left-most label, followed by the transport, followed by the base domain name.

The components of the authentication chain are built by starting at the target record set and its corresponding RRSIG. Then traversing the DNS tree upwards towards the trust anchor zone (normally the DNS root), for each zone cut, the DNSKEY and DS RRsets and their signatures are added. If DNS responses messages contain any domain names utilizing name compression [RFC1035], then they must be uncompressed.

In the future, proposed DNS protocol enhancements, such as the EDNS Chain Query extension [CHAINQUERY] may offer easy ways to obtain all of the chain data in one transaction with an upstream DNSSEC aware recursive server.

5. Caching and Regeneration of the Authentication Chain

DNS records have Time To Live (TTL) parameters, and DNSSEC signatures have validity periods (specifically signature expiration times). After the TLS server constructs the serialized authentication chain, it SHOULD cache and reuse it in multiple TLS connection handshakes. However, it MUST refresh and rebuild the chain as TTLs and signature validity periods dictate. A server implementation could carefully track these parameters and requery component records in the chain correspondingly. Alternatively, it could be configured to rebuild the entire chain at some predefined periodic interval that does not exceed the DNS TTLs or signature validity periods of the component records in the chain.

6. Verification

A TLS client making use of this specification, and which receives a DNSSEC authentication chain extension from a server, SHOULD use this information to perform DANE authentication of the server certificate. In order to do this, it uses the mechanism specified by the DNSSEC protocol [RFC4035]. This mechanism is sometimes implemented in a DNSSEC validation engine or library.

If the authentication chain is correctly verified, the client then performs DANE authentication of the server according to the DANE TLS protocol [RFC6698], and the additional protocol requirements outlined in [RFC7671].

7. Trust Anchor Maintenance

The trust anchor may change periodically, e.g. when the operator of the trust anchor zone performs a DNSSEC key rollover. Managed key rollovers typically use a process that can be tracked by verifiers allowing them to automatically update their trust anchors, as described in [RFC5011]. TLS clients using this specification are also expected to use such a mechanism to keep their trust anchors updated. Some operating systems may have a system-wide service to maintain and keep the root trust anchor up to date. In such cases, the TLS client application could simply reference that as its trust anchor, periodically checking whether it has changed.

8. Mandating use of this extension

A TLS server certificate MAY mandate the use of this extension by means of the X.509 TLS Feature Extension described in [RFC7633]. This X.509 certificate extension, when populated with the dnssec_chain TLS extension identifier, indicates to the client that the server must deliver the authentication chain when asked to do so.

(The X.509 TLS Feature Extension is the same mechanism used to deliver other mandatory signals, such as OCSP "must staple" assertions.)

9. Security Considerations

The security considerations of the normatively referenced RFCs (1035, 4034, 4035, 5246, 6066, 6698, 7633, 7671) all pertain to this extension. Since the server is delivering a chain of DNS records and signatures to the client, it MUST rebuild the chain in accordance with TTL and signature expiration of the chain components as described in Section 5. TLS clients need roughly accurate time in order to properly authenticate these signatures. This could be achieved by running a time synchronization protocol like NTP [RFC5905] or SNTP [RFC4330], which are already widely used today. TLS clients MUST support a mechanism to track and rollover the trust anchor key, or be able to avail themselves of a service that does this, as described in Section 7.

10. IANA Considerations

This extension requires the registration of a new value in the TLS ExtensionsType registry. The value requested from IANA is 53. If the draft is adopted by the WG, the authors expect to make an early allocation request as specified in [RFC7120].

11. Acknowledgments

Many thanks to Adam Langley for laying the groundwork for this extension. The original idea is his but our acknowledgment in no way implies his endorsement. This document also benefited from discussions with and review from the following people: Viktor Dukhovni, Daniel Kahn Gillmor, Jeff Hodges, Allison Mankin, Patrick McManus, Gowri Visweswaran, Duane Wessels, Nico Williams, and Paul Wouters.

12. References

12.1. Normative References

[RFC1035] Mockapetris, P., "Domain names - implementation and specification", STD 13, RFC 1035, November 1987.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

[RFC4034] Arends, R., Austein, R., Larson, M., Massey, D., and S. Rose, "Resource Records for the DNS Security Extensions", RFC 4034, March 2005.

[RFC4035] Arends, R., Austein, R., Larson, M., Massey, D., and S. Rose, "Protocol Modifications for the DNS Security Extensions", RFC 4035, March 2005.

[RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, August 2008.

[RFC6066] Eastlake, D., "Transport Layer Security (TLS) Extensions: Extension Definitions", RFC 6066, January 2011.

[RFC6698] Hoffman, P. and J. Schlyter, "The DNS-Based Authentication of Named Entities (DANE) Transport Layer Security (TLS) Protocol: TLSA", RFC 6698, August 2012.

[RFC7633] Hallam-Baker, P., "X.509v3 Transport Layer Security (TLS) Feature Extension", RFC 7633, DOI 10.17487/RFC7633, October 2015, <http://www.rfc-editor.org/info/rfc7633>.

[RFC7671] Dukhovni, V. and W. Hardaker, "The DNS-Based Authentication of Named Entities (DANE) Protocol: Updates and Operational Guidance", RFC 7671, DOI 10.17487/RFC7671, October 2015, <http://www.rfc-editor.org/info/rfc7671>.

12.2. Informative References

[RFC4330] Mills, D., "Simple Network Time Protocol (SNTP) Version 4 for IPv4, IPv6 and OSI", RFC 4330, January 2006.

[RFC5011] StJohns, M., "Automated Updates of DNS Security (DNSSEC) Trust Anchors", STD 74, RFC 5011, September 2007.

[RFC5905] Mills, D., Martin, J., Burbank, J., and W. Kasch, "Network Time Protocol Version 4: Protocol and Algorithms Specification", RFC 5905, June 2010.

[RFC7120] Cotton, M., "Early IANA Allocation of Standards Track Code Points", BCP 100, RFC 7120, January 2014.

[RFC7435] Dukhovni, V., "Opportunistic Security: Some Protection Most of the Time", RFC 7435, December 2014.

[RFC7672] Dukhovni, V. and W. Hardaker, "SMTP Security via Opportunistic DNS-Based Authentication of Named Entities (DANE) Transport Layer Security (TLS)", RFC 7672, DOI

10.17487/RFC7672, October 2015,
<http://www.rfc-editor.org/info/rfc7672>.

[AGL] Langley, A., "Serializing DNS Records with DNSSEC Authentication", , <https://tools.ietf.org/id/draft-agl-dane-serializechain-01.txt>.

[CHAINQUERY]
Wouters, P., "Chain Query Requests in DNS", , <https://tools.ietf.org/html/draft-ietf-dnsop-edns-chain-query>.

Appendix A. Pseudocode example

[code goes here]

Appendix B. Test vector

[data go here]

Authors' Addresses

Melinda Shore
No Mountain Software

EMail: melinda.shore@nomountain.net

Richard Barnes
Mozilla

EMail: rlb@ipv.sx

Shumon Huque
Verisign Labs

EMail: shuque@verisign.com

Willem Toorop
NLNet Labs

EMail: willem@nlnetlabs.nl