Internet Engineering Task Force (IETF)
Request for Comments: 8422
Obsoletes: 4492
Category: Standards Track
ISSN: 2070-1721

Y. Nir
Check Point
S. Josefsson
SJD AB
M. Pegourie-Gonnard
ARM

August 2018

Elliptic Curve Cryptography (ECC) Cipher Suites for Transport Layer Security (TLS) Versions 1.2 and Earlier

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 the Elliptic Curve Digital Signature Algorithm (ECDSA) and Edwards-curve Digital Signature Algorithm (EdDSA) as authentication mechanisms.

This document obsoletes RFC 4492.

Status of This Memo

This is an Internet Standards Track document.

This document is a product of the Internet Engineering Task Force (IETF). It represents the consensus of the IETF community. It has received public review and has been approved for publication by the Internet Engineering Steering Group (IESG). Further information on Internet Standards is available in Section 2 of RFC 7841.

Information about the current status of this document, any errata, and how to provide feedback on it may be obtained at https://www.rfc-editor.org/info/rfc8422.

Table of Contents

1. Introduction

This document describes additions to TLS to support ECC that are 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 [TLS1.3] and is explicitly out of scope for this document. In particular, this document defines:

- the use of the ECDHE key agreement scheme with ephemeral keys to establish the TLS premaster secret, and
- 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. 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", "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.

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.

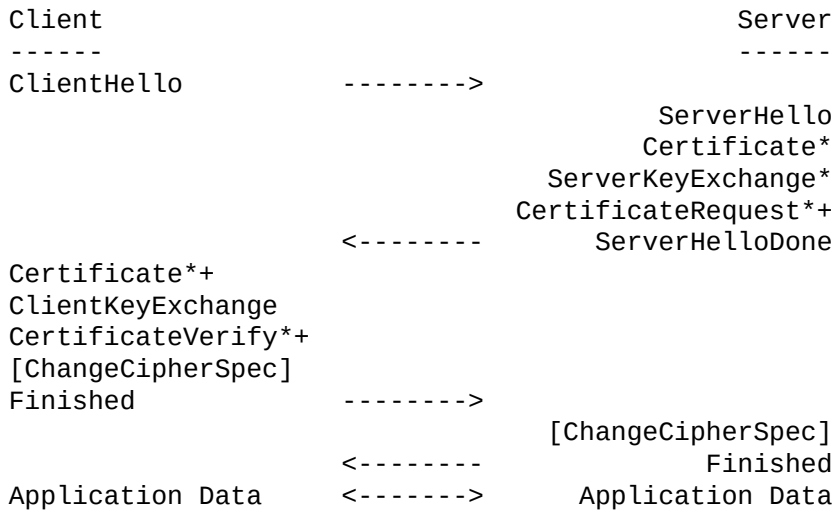

Figure 1: Message Flow in a Full TLS 1.2 Handshake

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 (see 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 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 is 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. Sections 5.3 and 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 "TLS 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 (now titled "TLS Supported Groups") 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 (now titled "TLS EC Point Formats") 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-prehashed) forms of EdDSA do not hash the data before signing it. For this reason, it does not make sense to combine them with a hash function in the extension.

For bits-on-the-wire compatibility with TLS 1.3, we define a new dummy value in the "TLS HashAlgorithm" registry that we call "Intrinsic" (value 8), meaning that hashing is intrinsic to the signature algorithm.

To represent ed25519 and ed448 in the signature_algorithms extension, the value shall be (8,7) and (8,8), 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. Sections 5.6 and 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 2 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 ECCurveType 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 (now titled "TLS EC Curve Types") 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;
```

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

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

- params: Specifies the ECDH public key and associated domain parameters.

- 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(7)
         ed448(8)
     } 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. SHA, in the above template for sha_hash, 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 [RFC8410] 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 include 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]. 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. ECC public keys must be encoded in certificates as described in Section 5.9. The certificate MUST contain an ECDSA- or EdDSA-capable public key.

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

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;
```

o 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 [X.690] corresponding to the following ASN.1 notation [X.680].

```
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 include 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 [RFC8410]. 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] using the ECKAS-DH1 scheme with the identity map as the 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 (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 Pseudorandom Function (PRF) serves as a KDF; in TLS 1.2, the KDF is determined by ciphersuite, and 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: (1) each party picks a secret key d uniformly at random and computes the corresponding public key x = X25519(d, G); (2) parties exchange their public keys and compute a shared secret as x_S = X25519(d, x_peer); and (3), 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 similarly, 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 runs 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 successors. Data to be signed/verified is run through the EdDSA algorithm with no hashing (EdDSA will internally run the data through the "prehash" function PH). 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 8 (Intrinsic).

All RSA signatures must be generated and verified according to Section 7.2 of [RFC8017].

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:

- TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256
- TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA
- TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256
- TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA

7. Implementation Status

Both ECDHE and ECDSA with the NIST curves are widely implemented and 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 they 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 Appendix D of each of the three TLS base documents apply accordingly.

Security discussions specific to ECC can be found in [IEEE.P1363] 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], [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, defined the IANA registries for the following:

- Supported Groups (Section 5.1)
- EC Point Format (Section 5.1)
- EC Curve Type (Section 5.4)

IANA has prepended "TLS" to the names of these 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". (RFC 4492 required IETF review.)

All existing entries in the "ExtensionType Values", "TLS ClientCertificateType Identifiers", "TLS Cipher Suites", "TLS Supported Groups", "TLS EC Point Format", and "TLS EC Curve Type" registries that referred to RFC 4492 have been updated to refer to this document.

IANA has assigned the value 29 to x25519 and the value 30 to x448 in the "TLS Supported Groups" registry.

IANA has assigned two values in the "TLS SignatureAlgorithm" registry for ed25519 (7) and ed448 (8) with this document as reference. This keeps compatibility with TLS 1.3.

IANA has assigned one value from the "TLS HashAlgorithm" registry for Intrinsic (8) with DTLS-OK set to true (Y) and this document as reference. This keeps compatibility with TLS 1.3.

10. References

10.1. 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, November 2005.

[FIPS.186-4]
National Institute of Standards and Technology, "Digital Signature Standard (DSS)", FIPS PUB 186-4, DOI 10.6028/NIST.FIPS.186-4, July 2013, <http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf>.

[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>.

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

[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, DOI 10.17487/RFC3279, April 2002, <https://www.rfc-editor.org/info/rfc3279>.

[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>.

[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>.

[RFC5289] Rescorla, E., "TLS Elliptic Curve Cipher Suites with SHA-256/384 and AES Galois Counter Mode (GCM)", RFC 5289, DOI 10.17487/RFC5289, August 2008, <https://www.rfc-editor.org/info/rfc5289>.

[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>.

[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>.

[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>.

[RFC8410] Josefsson, S. and J. Schaad, "Algorithm Identifiers for Ed25519, Ed448, X25519 and X448 for Use in the Internet X.509 Public Key Infrastructure", RFC 8410, DOI 10.17487/RFC8410, August 2018, <https://www.rfc-editor.org/info/rfc8410>.

[SECG-SEC2] Certicom Research, "SEC 2: Recommended Elliptic Curve Domain Parameters", Standards for Efficient Cryptography 2 (SEC 2), Version 2.0, January 2010, <http://www.secg.org/sec2-v2.pdf>.

[X.680] ITU-T, "Abstract Syntax Notation One (ASN.1): Specification of basic notation", ITU-T Recommendation X.680, ISO/IEC 8824-1, August 2015.

[X.690] ITU-T, "Information technology-ASN.1 encoding rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER)", ITU-T Recommendation X.690, ISO/IEC 8825-1, August 2015.

10.2. Informative References

[FIPS.180-4] National Institute of Standards and Technology, "Secure Hash Standard (SHS)", FIPS PUB 180-4, DOI 10.6028/NIST.FIPS.180-4, August 2015, <http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.180-4.pdf>.

[IEEE.P1363] IEEE, "Standard Specifications for Public Key Cryptography", IEEE Std P1363, <http://ieeexplore.ieee.org/document/891000/>.

[Menezes] Menezes, A. and B. Ustaoglu, "On reusing ephemeral keys in Diffie-Hellman key agreement protocols", International Journal of Applied Cryptography, Vol. 2, Issue 2, DOI 10.1504/IJACT.2010.038308, January 2010.

[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>.

[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>.

[TLS1.3] Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", Work in Progress, draft-ietf-tls-tls13-28, March 2018.

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. The following table displays the curve names chosen by different standards organizations; multiple names in one row represent aliases for the same curve.

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

- Renamed EllipticCurveList to NamedCurveList.
- 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.

Acknowledgements

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

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

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

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

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
ARM

Email: mpg@elzevir.fr