TLS Working Group
Internet-Draft
Obsoletes: 4492 (if approved)
Intended status: Standards Track
Expires: May 2, 2017

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

October 29, 2016

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

draft-ietf-tls-rfc4492bis-09

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 new authentication mechanisms.

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

Elliptic Curve Cryptography (ECC) has emerged as an attractive public-key cryptosystem, in particular for mobile (i.e., wireless) environments. Compared to currently prevalent cryptosystems such as RSA, ECC offers equivalent security with smaller key sizes. This is illustrated in the following table, based on [Lenstra_Verheul], which gives approximate comparable key sizes for symmetric- and asymmetric-key cryptosystems based on the best-known algorithms for attacking them.

Symmetric	ECC	DH/DSA/RSA
80	>=158	1024
112	>=221	2048
128	>=252	3072
192	>=379	7680
256	>=506	15360

Table 1: Comparable Key Sizes (in bits)

Smaller key sizes result in savings for power, memory, bandwidth, and computational cost that make ECC especially attractive for constrained environments.

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 Elliptic Curve Diffie-Hellman key agreement scheme with ephemeral keys to establish the TLS premaster secret, and
- o the use of ECDSA certificates 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 7 discusses security considerations. Section 8 describes IANA considerations for the name

spaces created by this document's predecessor. Section 9 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 2 summarizes the new key exchange algorithms. All of these key exchange algorithms provide forward secrecy.

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 2: 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. Implementations of this algorithm SHOULD provide authentication by other means.

Note that there is no structural difference between ECDH and ECDSA keys. A certificate issuer may use X.509 v3 keyUsage and extendedKeyUsage extensions to restrict the use of an ECC public key to certain computations. This document refers to an ECC key as ECDH-capable if its use in ECDH is permitted. ECDSA-capable and EdDSA-capable are defined similarly.

```
       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.

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.

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 should be sent as described in Section 2, and the CertificateVerify should 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 new 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 and compression techniques 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 new 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 EllipticCurveList. 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] and [CFRG-EdDSA]. The end result is as follows:

```
        enum {
            deprecated(1..22),
            secp256r1 (23), secp384r1 (24), secp521r1 (25),
            ecdh_x25519(29), ecdh_x448(30),
            eddsa_ed25519(TBD3), eddsa_ed448(TBD4),
            reserved (0xFE00..0xFEFF),
            deprecated(0xFF01..0xFF02),
            (0xFFFF)
        } NamedCurve;
```

Note that other specification have since added other values to this enumeration.

secp256r1, etc: Indicates support of the corresponding named curve or class of explicitly defined curves. 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 ecdh_x25519 and ecdh_x448 are defined in [RFC7748]. eddsa_ed25519 and eddsa_ed448 are signature-only curves defined in [CFRG-EdDSA]. Values 0xFE00 through 0xFEFF are reserved for private use.

The NamedCurve name space is maintained by IANA. See Section 8 for information on how new value assignments are added.

```
        struct {
            NamedCurve elliptic_curve_list<2..2^16-1>
        } EllipticCurveList;
```

Items in elliptic_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 MUST still be included, and contain exactly one value: the uncompressed point format (0).

The ECPointFormat name space is maintained by IANA. See Section 8 for information on how new value assignments are added.

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

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
```

5.2. 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 (e.g., the server may prefer to receive points in compressed format even when a client cannot parse this format: the same client may nevertheless be capable of outputting points in compressed format).

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 authorized for use in digital signatures.

Table 3: 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; in particular, the public key MUST employ a named curve (not the same curve as an explicit curve) unless the client has indicated support for explicit curves of the appropriate type. If the client has used a Supported Point Formats Extension, both the server's public key point and (in the case of an explicit curve) the curve's base point MUST respect the client's choice of point formats. (A server that cannot satisfy these requirements 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 SHOULD be used when applicable.

Values 248 through 255 are reserved for private use.

The ECCurveType name space is maintained by IANA. See Section 8 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;
```

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;
```

This identifies the type of the elliptic curve domain parameters.

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 values of NamedCurve except eddsa_ed25519(TBD3) and eddsa_ed448(TBD4).

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

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

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),
         eddsa(TBD5)
     } SignatureAlgorithm;
     select (SignatureAlgorithm) {
        case ecdsa:
             digitally-signed struct {
                 opaque sha_hash[sha_size];
             };
        case eddsa:
             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 [CFRG-EdDSA]. 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),
            rsa_fixed_ecdh(65),
            ecdsa_fixed_ecdh(66),
            (255)
        } ClientCertificateType;
```

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

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, RSA_fixed_ECDH, and ECDSA_fixed_ECDH authentication methods if the public key point specified in it respects the server's choice of point formats. If no Supported Point Formats Extension has been used, a certificate can only be considered suitable for use with these authentication methods if the point is represented in uncompressed point format.)

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).

Client Authentication Method	Client Certificate Type
ECDSA_sign	Certificate MUST contain an ECDSA- or EdDSA-capable public key.
ECDSA_fixed_ECDH	Certificate MUST contain an ECDH-capable public key on the same elliptic curve as the server's long-term ECDH key.
RSA_fixed_ECDH	The same as ECDSA_fixed_ECDH. The codepoints meant different things, but due to changes in TLS 1.2, both mean the same thing now.

Table 4: Client Certificate Types

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. If client authentication with ECDSA_fixed_ECDH or RSA_fixed_ECDH is used, this message is empty. Otherwise, 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"). (This is "explicit" in ECC cipher suites except when the client uses the ECDSA_fixed_ECDH or RSA_fixed_ECDH client authentication mechanism.)

```
        struct {
            select (PublicValueEncoding) {
                case implicit: struct { };
                case explicit: ECPoint ecdh_Yc;
            } ecdh_public;
        } 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 or compressed format. Curves eddsa_ed25519 and eddsa_ed448 MUST NOT be used here. Here, the format MUST conform to what the server has requested through a Supported Point Formats Extension if this extension was used, and MUST be uncompressed if this extension was not used.

```
        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 according to the ECKAS-DH1 scheme from IEEE 1363. It conveys this information to the client in the ClientKeyExchange message using the format defined above.

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, e.g., when the client is authenticated using the ECDSA_sign mechanism.

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 [CFRG-EdDSA]. 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 [CFRG-EdDSA].

EdDSA keys using Ed25519 and Ed25519ph algorithms MUST use the eddsa_ed25519 curve, and Ed448 and Ed448ph keys MUST use the eddsa_ed448 curve. Curves ecdh_x25519, ecdh_x448, eddsa_ed25519 and eddsa_ed448 MUST NOT be used for 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 ecdh_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 ecdh_x25519 with ecdh_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. The default hash function is SHA-1 [FIPS.180-2], and sha_size (see Section 5.4 and Section 5.8) is 20. However, an alternative hash function, such as one of the new SHA hash functions specified in FIPS 180-2 [FIPS.180-2], SHOULD be used instead.

All EdDSA computations MUST be performed according to [CFRG-EdDSA] 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).

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 0 (None).

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. 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.

X25519 was designed in a way that the result of X25519(x, d) will never reveal information about d, provided it was chosen as prescribed, for any value of x (the same holds true for X448).

All-zeroes output from X25519 or X448 MUST NOT be used for premaster secret (as required by definition of X25519 and X448). If the premaster secret would be all zeroes, the handshake MUST be aborted (most probably by sending a fatal alert).

Let's define legitimate values of x as the values that can be obtained as x = X25519(G, d') for some d', and call the other values illegitimate. The definition of the X25519 function shows that

legitimate values all share the following property: the high-order bit of the last byte is not set (for X448, any bit can be set).

Since there are some implementation of the X25519 function that impose this restriction on their input and others that don't, implementations of X25519 in TLS SHOULD reject public keys when the high-order bit of the final byte is set (in other words, when the value of the rightmost byte is greater than 0x7F) in order to prevent implementation fingerprinting. Note that this deviates from RFC 7748 which suggests that This value be masked.

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

Other than this recommended check, implementations do not need to ensure that the public keys they receive are legitimate: this is not necessary for security with X25519.

6. Cipher Suites

The table below defines new 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_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_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 5: 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].

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_ECDH_ECDSA_WITH_AES_128_CBC_SHA256

7. 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.

NEED TO ADD A PARAGRAPH HERE ABOUT WHY X25519/X448 ARE PREFERRABLE TO NIST CURVES.

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].

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

8. IANA Considerations

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

- Supported Groups Section 5.1
- ECPointFormat Section 5.1
- ECCurveType Section 5.4

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 assign two values from the NamedCurve registry with names eddsa_ed25519(TBD3) and eddsa_ed448(TBD4) with this document as reference. IANA has already assigned the value 29 to ecdh_x25519, and the value 30 to ecdh_x448.

IANA is requested to assign one value from SignatureAlgorithm Registry with name eddsa(TBD5) with this document as reference.

9. Acknowledgements

Most of the text is 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 author would like to thank Nikos Mavrogiannopoulos, Martin Thomson, and Tanja Lange for contributions to this document.

10. 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.

11. References

11.1. 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.

[CFRG-EdDSA]
Josefsson, S. and I. Liusvaara, "Edwards-curve Digital Signature Algorithm (EdDSA)", draft-irtf-cfrg-eddsa-08 (work in progress), August 2016.

[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.

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

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

11.2. Informative References

[FIPS.180-2] National Institute of Standards and Technology, "Secure Hash Standard", FIPS PUB 180-2, August 2002, <http://csrc.nist.gov/publications/fips/fips180-2/fips180-2.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.

[Lenstra_Verheul] Lenstra, A. and E. Verheul, "Selecting Cryptographic Key Sizes", Journal of Cryptology 14 (2001) 255-293, 2001.

[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.

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 6: 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