               Elliptic Curve Cryptography (ECC) Cipher Suites
                     for Transport Layer Security (TLS)

Status of This Memo

   This memo provides information for the Internet community. It does
   not specify an Internet standard of any kind. Distribution of this
   memo is unlimited.

Copyright Notice

Abstract

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

Table of Contents

1. Introduction

   Elliptic Curve Cryptography (ECC) is emerging 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 [18], 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	163	1024
112	233	2048
128	283	3072
192	409	7680
256	571	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
   both to TLS Version 1.0 [2] and to TLS Version 1.1 [3]. In
   particular, it defines

   o   the use of the Elliptic Curve Diffie-Hellman (ECDH) key agreement
       scheme with long-term or ephemeral keys to establish the TLS
       premaster secret, and

   o   the use of fixed-ECDH certificates and ECDSA 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 new 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. Section 9 gives acknowledgements.

This is followed by the lists of normative and informative references cited in this document, the authors' contact information, and statements on intellectual property rights and copyrights.

Implementation of this specification requires familiarity with TLS [2][3], TLS extensions [4], and ECC [5][6][7][11][17].

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 RFC 2119 [1].

2. Key Exchange Algorithms

This document introduces five new ECC-based key exchange algorithms for TLS. All of them use ECDH to compute the TLS premaster secret, and they differ only in the lifetime of ECDH keys (long-term or ephemeral) and 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.

The table below summarizes the new key exchange algorithms, which mimic DH_DSS, DHE_DSS, DH_RSA, DHE_RSA, and DH_anon (see [2] and [3]), respectively.

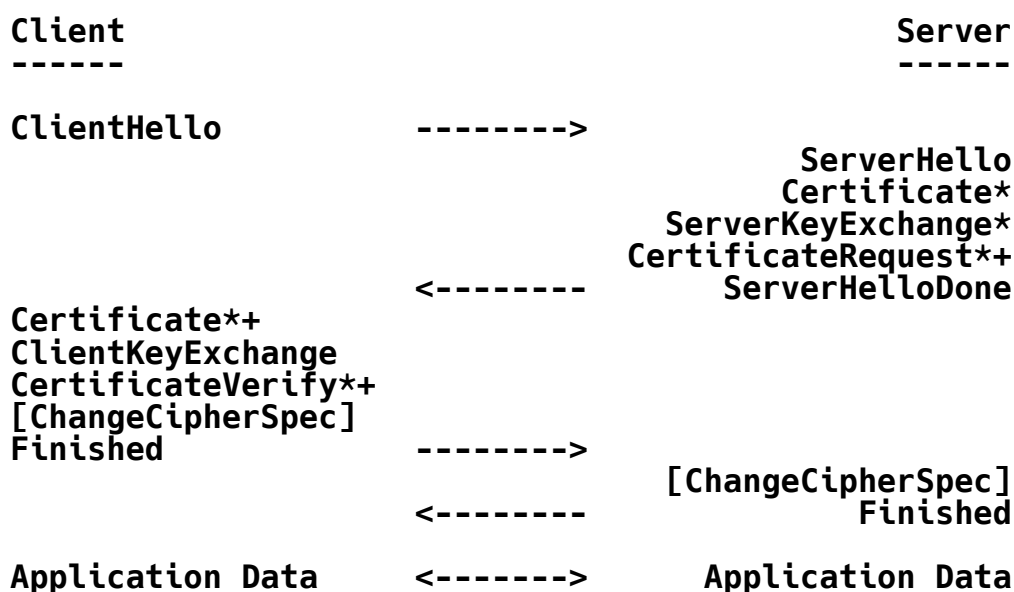| Key Exchange Algorithm	Description
ECDH_ECDSA	Fixed ECDH with ECDSA-signed certificates.
ECDHE_ECDSA	Ephemeral ECDH with ECDSA signatures.
ECDH_RSA	Fixed ECDH with RSA-signed certificates.
ECDHE_RSA	Ephemeral ECDH with RSA signatures.
ECDH_anon	Anonymous ECDH, no signatures.

Table 2: ECC Key Exchange Algorithms

The ECDHE_ECDSA and ECDHE_RSA key exchange mechanisms provide forward secrecy. 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 ECDH_RSA mechanism requires a server to acquire an ECC
certificate, but the certificate issuer can still use an existing RSA
key for signing. This eliminates the need to update the keys of
trusted certification authorities accepted by TLS clients. The
ECDH_ECDSA mechanism requires ECC keys for the server as well as the
certification authority and is best suited for constrained devices
unable to support RSA.

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 [15]. This document refers to an ECC key as
ECDH-capable if its use in ECDH is permitted. ECDSA-capable is
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 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 each 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. ECDH_ECDSA

In ECDH_ECDSA, the server's certificate MUST contain an ECDH-capable
public key and be signed with ECDSA.

A ServerKeyExchange MUST NOT be sent (the server's certificate
contains all the necessary keying information required by the client
to arrive at the premaster secret).

The client generates an ECDH key pair on the same curve as the
server's long-term public key and sends its public key in the
ClientKeyExchange message (except when using client authentication
algorithm ECDSA_fixed_ECDH or RSA_fixed_ECDH, in which case the
modifications from Section 3.2 or Section 3.3 apply).

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.

2.2. ECDHE_ECDSA

In ECDHE_ECDSA, the server's certificate MUST contain an ECDSA-
capable public key and be signed with ECDSA.

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 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.

2.3. ECDH_RSA

This key exchange algorithm is the same as ECDH_ECDSA except that the server's certificate MUST be signed with RSA rather than ECDSA.

2.4. 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. The server certificate MUST be signed with RSA.

2.5. ECDH_anon

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.

Note that while the ECDH_ECDSA, ECDHE_ECDSA, ECDH_RSA, and ECDHE_RSA key exchange algorithms require the server's certificate to be signed with a particular signature scheme, this specification (following the similar cases of DH_DSS, DHE_DSS, DH_RSA, and DHE_RSA in [2] and [3]) does not impose restrictions on signature schemes used elsewhere in the certificate chain. (Often such restrictions will be useful, and it is expected that this will be taken into account in certification authorities' signing practices. However, such restrictions are not strictly required in general: Even if it is beyond the capabilities of a client to completely validate a given chain, the client may be able to validate the server's certificate by relying on a trusted certification authority whose certificate appears as one of the intermediate certificates in the chain.)

3. Client Authentication

This document defines three new client authentication mechanisms,
each named after the type of client certificate involved: ECDSA_sign,
ECDSA_fixed_ECDH, and RSA_fixed_ECDH. 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 [2][3]. The ECDSA_fixed_ECDH and
RSA_fixed_ECDH mechanisms are usable with ECDH_ECDSA and ECDH_RSA.
Their use with ECDHE_ECDSA and ECDHE_RSA is prohibited because the
use of a long-term ECDH client key would jeopardize the forward
secrecy property of these algorithms.

The server can request ECC-based client authentication by including
one or more of these certificate types in its CertificateRequest
message. The server must not include any certificate types that are
prohibited for the negotiated key exchange algorithm. The client
must check if it possesses a certificate appropriate for any of the
methods 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-capable public key and signed with
ECDSA.

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.

3.2. ECDSA_fixed_ECDH

   To use this authentication mechanism, the client MUST possess a
   certificate containing an ECDH-capable public key, and that
   certificate MUST be signed with ECDSA. Furthermore, the client's
   ECDH key MUST be on the same elliptic curve as the server's long-term
   (certified) ECDH key. This might limit use of this mechanism to
   closed environments. In situations where the client has an ECC key
   on a different curve, it would have to authenticate using either
   ECDSA_sign or a non-ECC mechanism (e.g., RSA). Using fixed ECDH for
   both servers and clients is computationally more efficient than
   mechanisms providing forward secrecy.

   When using this authentication mechanism, the client MUST send an
   empty ClientKeyExchange as described in Section 5.7 and MUST NOT send
   the CertificateVerify message. The ClientKeyExchange is empty since
   the client's ECDH public key required by the server to compute the
   premaster secret is available inside the client's certificate. The
   client's ability to arrive at the same premaster secret as the server
   (demonstrated by a successful exchange of Finished messages) proves
   possession of the private key corresponding to the certified public
   key, and the CertificateVerify message is unnecessary.

3.3. RSA_fixed_ECDH

   This authentication mechanism is identical to ECDSA_fixed_ECDH except
   that the client's certificate MUST be signed with RSA.

   Note that while the ECDSA_sign, ECDSA_fixed_ECDH, and RSA_fixed_ECDH
   client authentication mechanisms require the client's certificate to
   be signed with a particular signature scheme, this specification does
   not impose restrictions on signature schemes used elsewhere in the
   certificate chain. (Often such restrictions will be useful, and it
   is expected that this will be taken into account in certification
   authorities' signing practices. However, such restrictions are not
   strictly required in general: Even if it is beyond the capabilities
   of a server to completely validate a given chain, the server may be
   able to validate the clients certificate by relying on a trust anchor
   that appears as one of the intermediate certificates in the chain.)

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 [4]; 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 Sections 2, 3, and 4. The
presentation language used here is the same as that used in TLS
[2][3]. 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 [4], 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 [4], 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. Sections 5.3 and 5.4).

   NOTE: A server participating in an ECDHE-ECDSA key exchange may use
   different curves for (i) the ECDSA key in its certificate, and (ii)
   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

```
enum {
    sect163k1 (1), sect163r1 (2), sect163r2 (3),
    sect193r1 (4), sect193r2 (5), sect233k1 (6),
    sect233r1 (7), sect239k1 (8), sect283k1 (9),
    sect283r1 (10), sect409k1 (11), sect409r1 (12),
    sect571k1 (13), sect571r1 (14), secp160k1 (15),
    secp160r1 (16), secp160r2 (17), secp192k1 (18),
    secp192r1 (19), secp224k1 (20), secp224r1 (21),
    secp256k1 (22), secp256r1 (23), secp384r1 (24),
    secp521r1 (25),
    reserved (0xFE00..0xFEFF),
    arbitrary_explicit_prime_curves(0xFF01),
    arbitrary_explicit_char2_curves(0xFF02),
    (0xFFFF)
} NamedCurve;
```

   sect163k1, etc: Indicates support of the corresponding named curve
      or class of explicitly defined curves. The named curves defined
      here are those specified in SEC 2 [13]. Note that many of these
      curves are also recommended in ANSI X9.62 [7] and FIPS 186-2 [11].
      Values 0xFE00 through 0xFEFF are reserved for private use. Values
      0xFF01 and 0xFF02 indicate that the client supports arbitrary
      prime and characteristic-2 curves, respectively (the curve
      parameters must be encoded explicitly in ECParameters).

   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<1..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 secp192r1 (aka NIST P-192;
value 19 = 0x0013) and secp224r1 (aka NIST P-224; value 21 = 0x0015)
and prefers to use secp192r1 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 13 00 15

A client that supports arbitrary explicit characteristic-2 curves
(value 0xFF02) would include an extension consisting of the following
octets:

     00 0A 00 04 00 02 FF 02

5.1.2. Supported Point Formats Extension

     enum { uncompressed (0), ansiX962_compressed_prime (1),
            ansiX962_compressed_char2 (2), reserved (248..255)
     } ECPointFormat;

     struct {
         ECPointFormat ec_point_format_list<1..2^8-1>
     } ECPointFormatList;

Three point formats are included in the definition of ECPointFormat
above. The uncompressed point format is the default format in that
implementations of this document MUST support it for all of their
supported curves. Compressed point formats reduce bandwidth by
including only the x-coordinate and a single bit of the y-coordinate
of the point. Implementations of this document MAY support the
ansiX962_compressed_prime and ansiX962_compressed_char2 formats,
where the former applies only to prime curves and the latter applies
only to characteristic-2 curves. (These formats are specified in
[7].) Values 248 through 255 are reserved for private use.

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 that can parse only the uncompressed point format (value 0) includes an extension consisting of the following octets; note that the first two octets indicate the extension type (Supported Point Formats Extension):

    00 0B 00 02 01 00

A client that in the case of prime fields prefers the compressed format (ansiX962_compressed_prime, value 1) over the uncompressed format (value 0), but in the case of characteristic-2 fields prefers the uncompressed format (value 0) over the compressed format (ansiX962_compressed_char2, value 2), may indicate these preferences by including an extension consisting of the following octets:

    00 0B 00 04 03 01 00 02

5.2. Server Hello Extension

This section specifies a TLS extension that can be included with the ServerHello message as described in [4], 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, or for the curve that is used in the server's public key that will appear in its Certificate message when using the ECDH_ECDSA or ECDH_RSA 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 elliptic_curve_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. 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 3 apply only to
the server's certificate (first in the chain).

Key Exchange Algorithm	Server Certificate Type
ECDH_ECDSA	Certificate MUST contain an ECDH-capable public key. It MUST be signed with ECDSA.
ECDHE_ECDSA	Certificate MUST contain an ECDSA-capable public key. It MUST be signed with ECDSA.
ECDH_RSA	Certificate MUST contain an ECDH-capable public key. It MUST be signed with RSA.
ECDHE_RSA	Certificate MUST contain an RSA public key authorized for use in digital signatures. It MUST be signed with RSA.

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.

Structure of this message:

        enum { explicit_prime (1), explicit_char2 (2),
               named_curve (3), reserved(248..255) } ECCurveType;

explicit_prime: Indicates the elliptic curve domain parameters are
    conveyed verbosely, and the underlying finite field is a prime
    field.

explicit_char2:   Indicates the elliptic curve domain parameters are
    conveyed verbosely, and the underlying finite field is a
    characteristic-2 field.

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.

        struct {
            opaque a <1..2^8-1>;
            opaque b <1..2^8-1>;
        } ECCurve;

   a, b: These parameters specify the coefficients of the elliptic
      curve. Each value contains the byte string representation of a
      field element following the conversion routine in Section 4.3.3 of
      ANSI X9.62 [7].

```
      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 [7]. This byte string may represent an elliptic curve point
      in uncompressed or compressed format; it MUST conform to what the
      client has requested through a Supported Point Formats Extension
      if this extension was used.

```
      enum { ec_basis_trinomial, ec_basis_pentanomial } ECBasisType;
```

   ec_basis_trinomial: Indicates representation of a characteristic-2
      field using a trinomial basis.

   ec_basis_pentanomial: Indicates representation of a
      characteristic-2 field using a pentanomial basis.

```
      struct {
          ECCurveType    curve_type;
          select (curve_type) {
              case explicit_prime:
                  opaque      prime_p <1..2^8-1>;
                  ECCurve     curve;
                  ECPoint     base;
                  opaque      order <1..2^8-1>;
                  opaque      cofactor <1..2^8-1>;
              case explicit_char2:
                  uint16      m;
                  ECBasisType basis;
                  select (basis) {
                      case ec_trinomial:
                          opaque  k <1..2^8-1>;
                      case ec_pentanomial:
                          opaque  k1 <1..2^8-1>;
                          opaque  k2 <1..2^8-1>;
                          opaque  k3 <1..2^8-1>;
                  };
                  ECCurve     curve;
                  ECPoint     base;
                  opaque      order <1..2^8-1>;
                  opaque      cofactor <1..2^8-1>;
```

```
            case named_curve:
                NamedCurve namedcurve;
        };
    } ECParameters;
```

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

prime_p: This is the odd prime defining the field Fp.

curve: Specifies the coefficients a and b of the elliptic curve E.

base: Specifies the base point G on the elliptic curve.

order: Specifies the order n of the base point.

cofactor: Specifies the cofactor $h = \#E(Fq)/n$, where $\#E(Fq)$
   represents the number of points on the elliptic curve E defined
   over the field Fq (either Fp or $F2^m$).

m: This is the degree of the characteristic-2 field $F2^m$.

k: The exponent k for the trinomial basis representation $x^m + x^k
   + 1$.

k1, k2, k3:   The exponents for the pentanomial representation $x^m +
   x^{k3} + x^{k2} + x^{k1} + 1$ (such that $k3 > k2 > k1$).

namedcurve: Specifies a recommended set of elliptic curve domain
   parameters. All those values of NamedCurve are allowed that refer
   to a specific curve. Values of NamedCurve that indicate support
   for a class of explicitly defined curves are not allowed here
   (they are only permissible in the ClientHello extension); this
   applies to arbitrary_explicit_prime_curves(0xFF01) and
   arbitrary_explicit_char2_curves(0xFF02).

```
    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 } SignatureAlgorithm;

select (SignatureAlgorithm) {
    case ecdsa:
        digitally-signed struct {
            opaque sha_hash[sha_size];
        };
} Signature;
```

```
ServerKeyExchange.signed_params.sha_hash
    SHA(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 [2][3]. SignatureAlgorithm is "ecdsa" 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 [9] corresponding to the
following ASN.1 notation [8].

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

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 [6]. 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.)

5.5. 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-capable public key and
                                  be signed with ECDSA.

          ECDSA_fixed_ECDH        Certificate MUST contain an
                                  ECDH-capable public key on the
                                  same elliptic curve as the server's
                                  long-term ECDH key.  This certificate
                                  MUST be signed with ECDSA.

          RSA_fixed_ECDH          Certificate MUST contain an
                                  ECDH-capable public key on the
                                  same elliptic curve as the server's
                                  long-term ECDH key.  This certificate
                                  MUST be signed with RSA.
```

                   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. 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 [6]. 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 [2] and [3], and the latter is extended here in
Section 5.4. For the ecdsa case, the signature field in the
CertificateVerify message contains an ECDSA signature computed over
handshake messages exchanged so far, exactly similar to
CertificateVerify with other signing algorithms in [2] and [3]:

        CertificateVerify.signature.sha_hash
            SHA(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 [9] corresponding to the following ASN.1
notation [8].

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

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 [14] or another RFC that replaces or extends it.
Clients SHOULD use the elliptic curve domain parameters recommended
in ANSI X9.62 [7], FIPS 186-2 [11], and SEC 2 [13].

5.10. ECDH, ECDSA, and RSA Computations

All ECDH calculations (including parameter and key generation as well
as the shared secret calculation) are performed according to [6]
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. As of TLS 1.0 [2] and
1.1 [3], this means that the MD5- and SHA-1-based TLS PRF serves as a
KDF; it is conceivable that future TLS versions or new TLS extensions
introduced in the future may vary this computation.)

All ECDSA computations MUST be performed according to ANSI X9.62 [7]
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 [10], and sha_size (see
Sections 5.4 and 5.8) is 20. However, an alternative hash function,
such as one of the new SHA hash functions specified in FIPS 180-2
[10], may be used instead if the certificate containing the EC public

   key explicitly requires use of another hash function. (The mechanism
   for specifying the required hash function has not been standardized,
   but this provision anticipates such standardization and obviates the
   need to update this document in response. Future PKIX RFCs may
   choose, for example, to specify the hash function to be used with a
   public key in the parameters field of subjectPublicKeyInfo.)

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

6. Cipher Suites

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

```
   CipherSuite TLS_ECDH_ECDSA_WITH_NULL_SHA          = { 0xC0, 0x01 }
   CipherSuite TLS_ECDH_ECDSA_WITH_RC4_128_SHA       = { 0xC0, 0x02 }
   CipherSuite TLS_ECDH_ECDSA_WITH_3DES_EDE_CBC_SHA  = { 0xC0, 0x03 }
   CipherSuite TLS_ECDH_ECDSA_WITH_AES_128_CBC_SHA   = { 0xC0, 0x04 }
   CipherSuite TLS_ECDH_ECDSA_WITH_AES_256_CBC_SHA   = { 0xC0, 0x05 }

   CipherSuite TLS_ECDHE_ECDSA_WITH_NULL_SHA         = { 0xC0, 0x06 }
   CipherSuite TLS_ECDHE_ECDSA_WITH_RC4_128_SHA      = { 0xC0, 0x07 }
   CipherSuite TLS_ECDHE_ECDSA_WITH_3DES_EDE_CBC_SHA = { 0xC0, 0x08 }
   CipherSuite TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA  = { 0xC0, 0x09 }
   CipherSuite TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA  = { 0xC0, 0x0A }

   CipherSuite TLS_ECDH_RSA_WITH_NULL_SHA            = { 0xC0, 0x0B }
   CipherSuite TLS_ECDH_RSA_WITH_RC4_128_SHA         = { 0xC0, 0x0C }
   CipherSuite TLS_ECDH_RSA_WITH_3DES_EDE_CBC_SHA    = { 0xC0, 0x0D }
   CipherSuite TLS_ECDH_RSA_WITH_AES_128_CBC_SHA     = { 0xC0, 0x0E }
   CipherSuite TLS_ECDH_RSA_WITH_AES_256_CBC_SHA     = { 0xC0, 0x0F }

   CipherSuite TLS_ECDHE_RSA_WITH_NULL_SHA           = { 0xC0, 0x10 }
   CipherSuite TLS_ECDHE_RSA_WITH_RC4_128_SHA        = { 0xC0, 0x11 }
   CipherSuite TLS_ECDHE_RSA_WITH_3DES_EDE_CBC_SHA   = { 0xC0, 0x12 }
   CipherSuite TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA    = { 0xC0, 0x13 }
   CipherSuite TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA    = { 0xC0, 0x14 }

   CipherSuite TLS_ECDH_anon_WITH_NULL_SHA           = { 0xC0, 0x15 }
   CipherSuite TLS_ECDH_anon_WITH_RC4_128_SHA        = { 0xC0, 0x16 }
   CipherSuite TLS_ECDH_anon_WITH_3DES_EDE_CBC_SHA   = { 0xC0, 0x17 }
   CipherSuite TLS_ECDH_anon_WITH_AES_128_CBC_SHA    = { 0xC0, 0x18 }
   CipherSuite 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 [2] and
[3]. AES ciphers are defined in [19].

Server implementations SHOULD support all of the following cipher
suites, and client implementations SHOULD support at least one of
them: TLS_ECDH_ECDSA_WITH_3DES_EDE_CBC_SHA,
TLS_ECDH_ECDSA_WITH_AES_128_CBC_SHA,
TLS_ECDHE_RSA_WITH_3DES_EDE_CBC_SHA, and
TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA.

7. Security Considerations

Security issues are discussed throughout this memo.

For TLS handshakes using ECC cipher suites, the security
considerations in appendices D.2 and D.3 of [2] and [3] apply
accordingly.

Security discussions specific to ECC can be found in [6] and [7].
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.

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 might be considered more conservative
than curves over F_2^m as there is no choice between multiple fields
of similar size for characteristic 2). Note, however, that algebraic
structure can also lead to implementation efficiencies, and
implementers and users may, therefore, need to balance conservatism
against a need for efficiency. Concrete attacks are known against
only very few special classes of curves, such as supersingular
curves, and these classes are excluded from the ECC standards that
this document references [6], [7].

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 [5], [6], [7], and [11].

Implementers and users must also consider whether they need forward
secrecy. Forward secrecy refers to the property that session keys
are not compromised if the static, certified keys belonging to the
server and client are compromised. The ECDHE_ECDSA and ECDHE_RSA key
exchange algorithms provide forward secrecy protection in the event
of server key compromise, while ECDH_ECDSA and ECDH_RSA do not.
Similarly, if the client is providing a static, certified key,
ECDSA_sign client authentication provides forward secrecy protection
in the event of client key compromise, while ECDSA_fixed_ECDH and
RSA_fixed_ECDH do not. Thus, to obtain complete forward secrecy
protection, ECDHE_ECDSA or ECDHE_RSA must be used for key exchange,
with ECDSA_sign used for client authentication if necessary. Here
again the security benefits of forward secrecy may need to be
balanced against the improved efficiency offered by other options.

8. IANA Considerations

This document describes three new name spaces for use with the TLS
protocol:

o NamedCurve (Section 5.1)

o ECPointFormat (Section 5.1)

o 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. Any
additional assignments require IETF Consensus action [16].

9. Acknowledgements

The authors wish to thank Bill Anderson and Tim Dierks.

10. References

10.1. Normative References

[1]   Bradner, S., "Key Words for Use in RFCs to Indicate Requirement
        Levels", RFC 2119, March 1997.

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

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

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

[5]   SECG, "Elliptic Curve Cryptography", SEC 1, 2000,
        <http://www.secg.org/>.

[6]   IEEE, "Standard Specifications for Public Key Cryptography",
        IEEE 1363, 2000.

[7]   ANSI, "Public Key Cryptography For The Financial Services
        Industry: The Elliptic Curve Digital Signature Algorithm
        (ECDSA)", ANSI X9.62, 1998.

[8]   International Telecommunication Union, "Information technology
        - Abstract Syntax Notation One (ASN.1): Specification of basic
        notation", ITU-T Recommendation X.680, 2002.

[9]   International Telecommunication Union, "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, 2002.

[10]   NIST, "Secure Hash Standard", FIPS 180-2, 2002.

[11]   NIST, "Digital Signature Standard", FIPS 186-2, 2000.

[12]   RSA Laboratories, "PKCS#1: RSA Encryption Standard version
        1.5", PKCS 1, November 1993.

[13]   SECG, "Recommended Elliptic Curve Domain Parameters", SEC 2,
        2000, <http://www.secg.org/>.

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

   [15] Housley, R., Polk, T., Ford, W., and D. Solo, "Internet X.509
         Public Key Infrastructure Certificate and Certificate
         Revocation List (CRL) Profile", RFC 3280, April 2002.

   [16] Narten, T. and H. Alvestrand, "Guidelines for Writing an IANA
         Considerations Section in RFCs", RFC 2434, October 1998.

10.2. Informative References

   [17] Harper, G., Menezes, A., and S. Vanstone, "Public-Key
         Cryptosystems with Very Small Key Lengths", Advances in
         Cryptology -- EUROCRYPT '92, LNCS 658, 1993.

   [18] Lenstra, A. and E. Verheul, "Selecting Cryptographic Key
         Sizes", Journal of Cryptology 14 (2001) 255-293,
         <http://www.cryptosavvy.com/>.

   [19] Chown, P., "Advanced Encryption Standard (AES) Ciphersuites for
         Transport Layer Security (TLS)", RFC 3268, June 2002.

Appendix A. Equivalent Curves (Informative)

   All of the NIST curves [11] and several of the ANSI curves [7] 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

Authors' Addresses

   Simon Blake-Wilson
   SafeNet Technologies BV
   Amstelveenseweg 88-90
   1075 XJ, Amsterdam
   NL

   Phone: +31 653 899 836
   EMail: sblakewilson@safenet-inc.com

   Nelson Bolyard
   Sun Microsystems Inc.
   4170 Network Circle
   MS SCA17-201
   Santa Clara, CA 95054
   US

   Phone: +1 408 930 1443
   EMail: nelson@bolyard.com

   Vipul Gupta
   Sun Microsystems Laboratories
   16 Network Circle
   MS UMPK16-160
   Menlo Park, CA 94025
   US

   Phone: +1 650 786 7551
   EMail: vipul.gupta@sun.com

   Chris Hawk
   Corriente Networks LLC
   1563 Solano Ave., #484
   Berkeley, CA 94707
   US

   Phone: +1 510 527 0601
   EMail: chris@corriente.net

   Bodo Moeller
   Ruhr-Uni Bochum
   Horst-Goertz-Institut, Lehrstuhl fuer Kommunikationssicherheit
   IC 4/139
   44780 Bochum
   DE

   Phone: +49 234 32 26795
   EMail: bodo@openssl.org

Full Copyright Statement

   Copyright (C) The Internet Society (2006).

   This document is subject to the rights, licenses and restrictions
   contained in BCP 78, and except as set forth therein, the authors
   retain all their rights.

   This document and the information contained herein are provided on an
   "AS IS" basis and THE CONTRIBUTOR, THE ORGANIZATION HE/SHE REPRESENTS
   OR IS SPONSORED BY (IF ANY), THE INTERNET SOCIETY AND THE INTERNET
   ENGINEERING TASK FORCE DISCLAIM ALL WARRANTIES, EXPRESS OR IMPLIED,
   INCLUDING BUT NOT LIMITED TO ANY WARRANTY THAT THE USE OF THE
   INFORMATION HEREIN WILL NOT INFRINGE ANY RIGHTS OR ANY IMPLIED
   WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

Intellectual Property

   The IETF takes no position regarding the validity or scope of any
   Intellectual Property Rights or other rights that might be claimed to
   pertain to the implementation or use of the technology described in
   this document or the extent to which any license under such rights
   might or might not be available; nor does it represent that it has
   made any independent effort to identify any such rights. Information
   on the procedures with respect to rights in RFC documents can be
   found in BCP 78 and BCP 79.

   Copies of IPR disclosures made to the IETF Secretariat and any
   assurances of licenses to be made available, or the result of an
   attempt made to obtain a general license or permission for the use of
   such proprietary rights by implementers or users of this
   specification can be obtained from the IETF on-line IPR repository at
   http://www.ietf.org/ipr.

   The IETF invites any interested party to bring to its attention any
   copyrights, patents or patent applications, or other proprietary
   rights that may cover technology that may be required to implement
   this standard. Please address the information to the IETF at
   ietf-ipr@ietf.org.

Acknowledgement

   Funding for the RFC Editor function is provided by the IETF
   Administrative Support Activity (IASA).