TLS Working Group V. Gupta
Internet-Draft Sun Labs
Expires: September 2, 2005 S. Blake-Wilson
BCI
B. Moeller
University of Calgary
C. Hawk
Corriente Networks
N. Bolyard
Mar. 2005

ECC Cipher Suites for TLS
<draft-ietf-tls-ecc-08.txt>

Status of this Memo

This Internet-Draft will expire on September 2, 2005.

Copyright Notice

Abstract

This document describes new key exchange algorithms based on Elliptic Curve Cryptography (ECC) for the TLS (Transport Layer Security) 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.

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].

Please send comments on this document to the TLS mailing list.

Table of Contents

1. Introduction

Elliptic Curve Cryptography (ECC) is emerging as an attractive public-key cryptosystem for mobile/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 [12], 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)

Figure 1

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

This document describes additions to TLS to support ECC. In particular, it defines

- 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
- 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 and Section 8 mention security considerations and acknowledgments, respectively. This is followed by a list of 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], TLS extensions [3] and ECC [4][5][6][8] .

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 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, DH_RSA, DHE_DSS, DHE_RSA and DH_anon (see [2]), 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

Figure 2

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 trusted key store in 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 X509.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 is defined similarly.

```
              Client                                        Server
              ------                                        ------

              ClientHello          -------->
                                                       ServerHello
                                                      Certificate*
                                                ServerKeyExchange*
                                              CertificateRequest*+
                                   <--------       ServerHelloDone
              Certificate*+
              ClientKeyExchange
              CertificateVerify*+
              [ChangeCipherSpec]
              Finished             -------->
                                                [ChangeCipherSpec]
                                   <--------              Finished

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

Figure 1: Message flow in a full TLS handshake
* message is not sent under some conditions
+ message is not sent unless the client is
authenticated

Figure 3

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 MUST generate an ECDH key pair on the same curve as the server's long-term public key and send 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 Section 3.2 or Section 3.3 apply).

Both client and server MUST 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 MUST send 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 MUST generate an ECDH key pair on the same curve as the server's ephemeral ECDH key and send its public key in the ClientKeyExchange message.

Both client and server MUST 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 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 the server's certificate MUST contain an RSA public key authorized for signing and the signature in the ServerKeyExchange message MUST be computed with the corresponding RSA private key. 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 MUST generate an ECDH key pair on the same curve as the server's ephemeral ECDH key and send its public key in the ClientKeyExchange message.

Both client and server MUST 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

3. Client Authentication

This document defines three new client authentication mechanisms 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]. 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 MUST prove 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 either using 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 the client's certificate MUST be signed with RSA.

4. TLS Extensions for ECC

Two new TLS extensions --- (i) the Supported Elliptic Curves Extension, and (ii) the Supported Point Formats Extension --- allow a client to negotiate the use of specific curves and point formats (e.g. compressed v/s uncompressed), respectively. 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 [3]. The client enumerates the curves and point formats it supports by including the appropriate extensions in its ClientHello message. By echoing that extension in its ServerHello, the server agrees to restrict its key selection or encoding to the choices specified by the client.

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 negotiate the use of an ECC cipher suite only if they can complete the handshake while limiting themselves to the curves and compression techniques enumerated 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.

These extensions MUST NOT be included if the client does not propose any ECC cipher suites. A client that proposes ECC cipher suites may choose not to include these extension. 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.

5. Data Structures and Computations

This section specifies the data structures and computations used by ECC-based key mechanisms specified in Section 2, Section 3 and Section 4. The presentation language used here is the same as that used in TLS [2]. 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

When this message is sent:

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

Meaning of this message:

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

Structure of this message:

The general structure of TLS extensions is described in [3] and this specification adds two new types to ExtensionType.

```
     enum { elliptic_curves(??), ec_point_formats(??) } ExtensionType;
```

elliptic_curves: Indicates the set of elliptic curves supported by the client. For this extension, the opaque extension_data field contains EllipticCurveList.

ec_point_formats: Indicates the set of point formats supported by the client. For this extension, the opaque extension_data field contains ECPointFormatList.

```
        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 (240..247),
            arbitrary_explicit_prime_curves(253),
            arbitrary_explicit_char2_curves(254),
            (255)
        } NamedCurve;
```

sect163k1, etc: Indicates support of the corresponding named curve specified in SEC 2 [10]. Note that many of these curves are also recommended in ANSI X9.62 [6], and FIPS 186-2 [8]. Values 240 through 247 are reserved for private use. Values 253 and 254 indicate that the client supports arbitrary prime and characteristic-2 curves, respectively (the curve parameters must be encoded explicitly in ECParameters).

```
        struct {
            NamedCurve elliptic_curve_list<1..2^8-1>
        } 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) and secp224r1 (aka NIST P-224) and prefers to use secp192r1, would include an elliptic_curves extension with the following octets:

```
        00 ?? 00 03 02 13 15
```

A client that supports arbitrary explicit binary polynomial curves would include an extension with the following octets:

```
        00 ?? 00 02 01 fe
```

```
        enum { uncompressed (0), ansiX963_compressed (1),
               ansiX963_hybrid (2), (255)
        } ECPointFormat;

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

Three point formats are included in the defintion of ECPointFormat above. The uncompressed point format is the default format that implementations of this document MUST support. The ansix963_compressed format reduces bandwidth by including only the x-coordinate and a single bit of the y-coordinate of the point. The ansix963_hybrid format includes both the full y-coordinate and the compressed y-coordinate to allow flexibility and improve efficiency in some cases. Implementations of this document MAY support the ansix963_compressed and ansix963_hybrid point formats.

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

A client that only supports the uncompressed point format includes an extension with the following octets:

```
        00 ?? 00 02 01 00
```

A client that prefers the use of the ansiX963_compressed format over uncompressed may indicate that preference by including an extension with the following octets:

```
        00 ?? 00 03 02 01 00
```

Actions of the sender:

A client that proposes ECC cipher suites in its ClientHello appends these extensions (along with any others) enumerating the curves and point formats it supports.

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.

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 "elliptic_curves" extension in selecting both of these curves.

If a server does not understand the "elliptic_curves" extension or is unable to complete the ECC handshake while restricting itself to the enumerated curves, 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.

5.2 Server Hello Extensions

When this message is sent:

The ServerHello ECC extensions are sent in response to a Client Hello message containing ECC extensions when negotiating an ECC cipher suite.

Meaning of this message:

These extensions indicate the server's agreement to use only the elliptic curves and point formats supported by the client during the ECC-based key exchange.

Structure of this message:

The ECC extensions echoed by the server are the same as those in the ClientHello except the "extension_data" field is empty.

For example, a server indicates its acceptance of the client's elliptic_curves extension by sending an extension with the following octets:

```
     00 ?? 00 00
```

Actions of the sender:

A server makes sure that it can complete a proposed ECC key exchange mechanism by restricting itself to the curves/point formats supported by the client before sending these extensions.

Actions of the receiver:

A client that receives a ServerHello with ECC extensions proceeds with an ECC key exchange assured that it will be able to handle the server's EC key(s).

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.

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.

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), (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.

```
     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 [6].

```
     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 [6]. Note that this byte string may represent an elliptic curve point in compressed or uncompressed form.

```
        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.

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 enum values of NamedCurve are allowed except for arbitrary_explicit_prime_curves(253) and arbitrary_explicit_char2_curves(254). These two values are only allowed in the ClientHello extension.

```
     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;
```

NOTE: SignatureAlgorithm is 'rsa' for the ECDHE_RSA key exchange algorithm and 'anonymous' for ECDH_anon. These cases are defined in TLS [2]. SignatureAlgorithm is 'ecdsa' for ECDHE_ECDSA. ECDSA signatures are generated and verified as described in Section 5.10. As per ANSI X9.62, an ECDSA signature consists of a pair of integers r and s. These integers are both converted into byte strings of the same length as the curve order n using the conversion routine specified in Section 4.3.1 of [6]. The two byte strings are concatenated, and the result is placed in the signature field.

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 [5]. It conveys this information to the client in the ServerKeyExchange message using the format defined above.

Actions of the recipient:

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.

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(?), rsa_fixed_ecdh(?),
```

```
            ecdsa_fixed_ecdh(?), (255)
        } ClientCertificateType;
```

ecdsa_sign, etc Indicates that the server would like to use the corresponding client authentication method specified in Section 3. EDITOR: The values used for ecdsa_sign, rsa_fixed_ecdh, and ecdsa_fixed_ecdh have been left as ?. These values will be assigned when this draft progresses to RFC. Earlier versions of this draft used the values 5, 6, and 7 - however these values have been removed since they are used differently by SSL 3.0 [13] and their use by TLS is being deprecated.

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 an appropriate certificate for use with any of the requested methods, and decides whether or not 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.

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 { yes, no } EphemeralPublicKey;
```

yes, no: Indicates whether or not the client is providing an ephemeral ECDH public key. (In ECC ciphersuites, this is "yes" except when the client uses the ECDSA_fixed_ECDH or RSA_fixed_ECDH client authentication mechanism.)

```
     struct {
         select (EphemeralPublicKey) {
             case yes: ECPoint  ecdh_Yc;
             case no:  struct { };
         } ecdh_public;
     } ClientECDiffieHellmanPublic;
```

ecdh_Yc: Contains the client's ephemeral ECDH public key.

```
     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 [5]. It conveys this information to the client in the ClientKeyExchange message using the format defined above.

Actions of the recipient:

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 is extended as follows.

```
     enum { ecdsa } SignatureAlgorithm;

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

For the ecdsa case, the signature field in the CertificateVerify message contains an ECDSA signature computed over handshake messages exchanged so far. ECDSA signatures are computed as described in Section 5.10. As per ANSI X9.62, an ECDSA signature consists of a pair of integers r and s. These integers are both converted into byte strings of the same length as the curve order n using the conversion routine specified in Section 4.3.1 of [6]. The two byte strings are concatenated, and the result is placed in the signature field.

Actions of the sender:

The client computes its signature over all handshake messages sent or received starting at client hello 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

X509 certificates containing ECC public keys or signed using ECDSA MUST comply with [11] or another RFC that replaces or extends it. Clients SHOULD use the elliptic curve domain parameters recommended in ANSI X9.62 [6], FIPS 186-2 [8], and SEC 2 [10].

5.10 ECDH, ECDSA and RSA Computations

All ECDH calculations (including parameter and key generation as well as the shared secret calculation) MUST be performed according to [5] using the ECKAS-DH1 scheme with the identity map as key derivation function, so that the premaster secret is the x-coordinate of the ECDH shared secret elliptic curve point, i.e. the octet string Z in IEEE 1363 terminology.

Note that a new extension may be introduced in the future to allow the use of a different KDF during computation of the premaster secret. In this event, the new KDF would be used in place of the process detailed above. This may be desirable, for example, to support compatibility with the planned NIST key agreement standard.

All ECDSA computations MUST be performed according to ANSI X9.62 [6] 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 [7] 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 [7], 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 [9].

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           = { 0x00, 0x?? }
  CipherSuite TLS_ECDH_ECDSA_WITH_RC4_128_SHA        = { 0x00, 0x?? }
  CipherSuite TLS_ECDH_ECDSA_WITH_DES_CBC_SHA        = { 0x00, 0x?? }
  CipherSuite TLS_ECDH_ECDSA_WITH_3DES_EDE_CBC_SHA   = { 0x00, 0x?? }
  CipherSuite TLS_ECDH_ECDSA_WITH_AES_128_CBC_SHA    = { 0x00, 0x?? }
  CipherSuite TLS_ECDH_ECDSA_WITH_AES_256_CBC_SHA    = { 0x00, 0x?? }

  CipherSuite TLS_ECDHE_ECDSA_WITH_NULL_SHA          = { 0x00, 0x?? }
  CipherSuite TLS_ECDHE_ECDSA_WITH_RC4_128_SHA       = { 0x00, 0x?? }
  CipherSuite TLS_ECDHE_ECDSA_WITH_3DES_EDE_CBC_SHA  = { 0x00, 0x?? }
  CipherSuite TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA   = { 0x00, 0x?? }
  CipherSuite TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA   = { 0x00, 0x?? }

  CipherSuite TLS_ECDH_RSA_WITH_NULL_SHA             = { 0x00, 0x?? }
  CipherSuite TLS_ECDH_RSA_WITH_RC4_128_SHA          = { 0x00, 0x?? }
  CipherSuite TLS_ECDH_RSA_WITH_3DES_EDE_CBC_SHA     = { 0x00, 0x?? }
  CipherSuite TLS_ECDH_RSA_WITH_AES_128_CBC_SHA      = { 0x00, 0x?? }
  CipherSuite TLS_ECDH_RSA_WITH_AES_256_CBC_SHA      = { 0x00, 0x?? }

  CipherSuite TLS_ECDHE_RSA_WITH_NULL_SHA            = { 0x00, 0x?? }
  CipherSuite TLS_ECDHE_RSA_WITH_RC4_128_SHA         = { 0x00, 0x?? }
  CipherSuite TLS_ECDHE_RSA_WITH_3DES_EDE_CBC_SHA    = { 0x00, 0x?? }
  CipherSuite TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA     = { 0x00, 0x?? }
  CipherSuite TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA     = { 0x00, 0x?? }

  CipherSuite TLS_ECDH_anon_NULL_WITH_SHA            = { 0x00, 0x?? }
  CipherSuite TLS_ECDH_anon_WITH_RC4_128_SHA         = { 0x00, 0x?? }
  CipherSuite TLS_ECDH_anon_WITH_3DES_EDE_CBC_SHA    = { 0x00, 0x?? }
  CipherSuite TLS_ECDH_anon_WITH_AES_128_CBC_SHA     = { 0x00, 0x?? }
  CipherSuite TLS_ECDH_anon_WITH_AES_256_CBC_SHA     = { 0x00, 0x?? }
```

Table 5: TLS ECC cipher suites

Figure 30

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]. AES ciphers are defined in [14].

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

This document is based on [2], [5], [6] and [14]. The appropriate security considerations of those documents apply.

One important issue that implementors and users must consider is elliptic curve selection. Guidance on selecting an appropriate elliptic curve size is given in Figure 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 implementors 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 [5], [6].

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 [4], [5], [6], [8].

Implementors 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. Acknowledgments

The authors wish to thank Bill Anderson and Tim Dierks.

9. References

9.1 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] Blake-Wilson, S., Nystrom, M., Hopwood, D., Mikkelsen, J. and T. Wright, "Transport Layer Security (TLS) Extensions", Internet-draft draft-ietf-tls-rfc3546bis-00.txt, Nov. 2004.

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

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

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

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

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

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

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

[11] 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.

9.2 Informative References

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

[13] Freier, A., Karlton, P. and P. Kocher, "The SSL Protocol Version 3.0", November 1996, <http://wp.netscape.com/eng/ssl3/draft302.txt>.

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

[15] Hovey, R. and S. Bradner, "The Organizations Involved in the IETF Standards Process", RFC 2028, BCP 11, October 1996.

Authors' Addresses

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

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

Simon Blake-Wilson
Basic Commerce & Industries, Inc.
96 Spandia Ave
Unit 606
Toronto, ON M6G 2T6
Canada

Phone: +1 416 214 5961
Email: sblakewilson@bcisse.com

Bodo Moeller
University of Calgary
Dept of Math & Stats
2500 University Dr NW
Calgary, AB T2N 1N4
CA

Email: bodo@openssl.org

Chris Hawk
Corriente Networks

Email: chris@corriente.net

Nelson Bolyard

Email: nelson@bolyard.com

Intellectual Property Statement

Disclaimer of Validity

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.

Copyright Statement

Acknowledgment

Funding for the RFC Editor function is currently provided by the Internet Society.