TLS Working Group
Internet-Draft
Expires: February 27, 2003

V. Gupta
Sun Labs
S. Blake-Wilson
BCI
B. Moeller
Technische Universitaet Darmstadt
C. Hawk
Independent Consultant

August 29, 2002

ECC Cipher Suites for TLS

<draft-ietf-tls-ecc-02.txt>

Status of this Memo

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 [2], 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
128	283	3072
192	409	7680
256	571	15360

Table 1: Comparable key sizes (in bits)

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. Section 4 specifies various data structures needed for an ECC-based handshake, their encoding in TLS messages and the processing of those messages. Section 5 defines new ECC-based cipher suites and identifies a small subset of these as recommended for all implementations of this specification. Section 6, Section 7 and Section 8 mention security considerations, intellectual property rights, and acknowledgments, respectively. This is followed by a list of references cited in this document and the authors' contact information.

Implementation of this specification requires familiarity with both

TLS [3] and ECC [5][6][7][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 [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

Note that 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 1 shows all messages involved in the TLS key establishment protocol (aka full handshake). The addition of ECC has direct impact only on the 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.

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 4.8

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 4.8) 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 4.8

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 [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 4.4) 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 4.6.

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.

When using this authentication mechanism, the client MUST send an empty ClientKeyExchange as described in Section 4.5 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. Data Structures and Computations

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

4.1 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 4.7.

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.

4.2 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>;
    opaque seed <0..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].

seed: This is an optional parameter used to derive the coefficients of a randomly generated elliptic curve.

```
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]. Note that this byte string may represent an elliptic curve point in compressed or uncompressed form. Implementations of this specification MUST support the uncompressed form and MAY support the compressed form.


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

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

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

```
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), (255)
} NamedCurve;
```

sect163k1, etc: Indicates use of the corresponding named curve specified in SEC 2 [12]. Note that many of these curves are also recommended in ANSI X9.62 [7], and FIPS 186-2 [9]. Values 240 through 247 are reserved for private use.


```
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 (and optional seed) 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-two 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.

```
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[20];
                };
        } Signature;
```

NOTE: SignatureAlgorithm is 'rsa' for the ECDHE_RSA key exchange algorithm and 'anonymous' for ECDH_anon. These cases are defined in TLS [3]. SignatureAlgorithm is 'ecdsa' for ECDHE_ECDSA. ECDSA signatures are generated and verified as described in Section 4.8.

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 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.

4.3 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(5), rsa_fixed_ecdh(6),
    ecdsa_fixed_ecdh(7), (255)
} ClientCertificateType;
```

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

NOTE: SSL 3.0 [4] assigns values 5 and 6 differently (rsa_ephemeral_dh and dss_ephemeral_dh); these ClientCertificateType values are not used by TLS.

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.

4.4 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 4.7.

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.

4.5 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 [6]. 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.

4.6 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[20];
        };
} 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 4.8. 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 [7]. 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.

4.7 Elliptic Curve Certificates

X509 certificates containing ECC public keys or signed using ECDSA MUST comply with [14]. Clients SHOULD use the elliptic curve domain parameters recommended in ANSI X9.62 [7], FIPS 186-2 [9], and SEC 2 [12].

4.8 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 [6] using the ECKAS-DH1 scheme with the ECSVDP-DH secret value derivation primitive, and the KDF1 key derivation function using SHA-1 [9]. The output of this scheme, i.e. the 20-byte SHA-1 output from the KDF, is the premaster secret.

DISCUSSION POINT:

Using KDF1 with SHA-1 limits the security to at most 160 bits, independently of the elliptic curve used for ECDH. An alternative way to define the protocol would be to employ the identity map as key derivation function (in other words, omit the SHA-1 step and directly use the octet string representation of the x coordinate of the elliptic curve point resulting from the ECDH computation as premaster secret). This is similar to conventional DH in TLS [3], and it is appropriate for TLS given that TLS already defines a PRF for determining the actual symmetric keys.

The TLS PRF (which is used to derive the master secret from the premaster secret and the symmetric keys from the master secret) has an internal security limit of at most 288 bits (128 + 160 for MD5 and SHA-1), so the use of KDF1 with SHA-1 can be seen to actually weaken the theoretical security of the protocol.

Options to solve this problem include the following:

- o Omit the SHA-1 step as describe above. (BM)

- o Continue to use KDF1 with SHA-1 for curves up to, say, 288 bits (more precisely: for curves where FE2OSP returns an octet string up to 36 octets) and omit the SHA-1 step only for larger curves. (SBW/BM)

- o Continue to use KDF1 with SHA-1 for curves up to, say, 288 bits and use KDF1 with SHA-256 or SHA-384 or SHA-512 for larger curves. (SBW)

The first solution will break compatibility with existing implementations based on draft-ietf-tls-ecc-01.txt. The second and third solutions are somewhat of a kludge, but maintain compatibility (in a large range).

OPEN QUESTION: We invite comments on which of these solutions would be preferred.

END OF DISCUSSION POINT.

All ECDSA computations MUST be performed according to ANSI X9.62 [7] using the SHA-1 [9] hash function. The 20 bytes of the SHA-1 are run directly through the ECDSA algorithm with no additional hashing.

All RSA signatures must be generated and verified according to PKCS#1 [10].

5. 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, 0x47 }
CipherSuite TLS_ECDH_ECDSA_WITH_RC4_128_SHA        = { 0x00, 0x48 }
CipherSuite TLS_ECDH_ECDSA_WITH_DES_CBC_SHA        = { 0x00, 0x49 }
CipherSuite TLS_ECDH_ECDSA_WITH_3DES_EDE_CBC_SHA   = { 0x00, 0x4A }
CipherSuite TLS_ECDH_ECDSA_WITH_AES_128_CBC_SHA    = { 0x00, 0x4B }
CipherSuite TLS_ECDH_ECDSA_WITH_AES_256_CBC_SHA    = { 0x00, 0x4C }

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

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 [3]. AES ciphers are defined in [11].

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.

6. Security Considerations

This document is entirely concerned with security mechanisms.

This document is based on [3], [6], [7] and [11]. The appropriate security considerations of those documents apply.

7. Intellectual Property Rights

The IETF has been notified of intellectual property rights claimed in regard to the specification contained in this document. For more information, consult the online list of claimed rights (http://www.ietf.org/ipr.html).

The IETF takes no position regarding the validity or scope of any intellectual property 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; neither does it represent that it has made any effort to identify any such rights. Information on the IETF's procedures with respect to rights in standards-track and standards-related documentation can be found in [13]. Copies of claims of rights made available for publication 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 Secretariat.

8. Acknowledgments

The authors wish to thank Bill Anderson and Tim Dierks.

References

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

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

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

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

[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] NIST, "Digital Signature Standard", FIPS 180-1, 2000.

[9] NIST, "Secure Hash Standard", FIPS 186-2, 1995.

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

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

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

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

[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.

Authors' Addresses

Vipul Gupta
Sun Microsystems Laboratories
2600 Casey Avenue
MS UMTV29-235
Mountain View, CA 94303
USA

Phone: +1 650 336 1681
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
Technische Universitaet Darmstadt
Alexanderstr. 10
64283 Darmstadt
Germany

Phone: +49 6151 16 6628
EMail: moeller@cdc.informatik.tu-darmstadt.de

Chris Hawk
Independent Consultant

EMail: chris@socialeng.com

Full Copyright Statement

Acknowledgement

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