Tim Dierks
Independent
Eric Rescorla
Network Resonance, Inc.

INTERNET-DRAFT
<draft-ietf-tls-rfc4346-bis-01.txt>
June 2006 (Expires December 2006)

The TLS Protocol
Version 1.2

Status of this Memo

Copyright Notice

Abstract

This document specifies Version 1.2 of the Transport Layer Security (TLS) protocol. The TLS protocol provides communications security over the Internet. The protocol allows client/server applications to communicate in a way that is designed to prevent eavesdropping, tampering, or message forgery.

Table of Contents

Change history

18-Feb-06   First draft by ekr@rtfm.com

1. Introduction

The primary goal of the TLS Protocol is to provide privacy and data integrity between two communicating applications. The protocol is composed of two layers: the TLS Record Protocol and the TLS Handshake Protocol. At the lowest level, layered on top of some reliable transport protocol (e.g., TCP[TCP]), is the TLS Record Protocol. The TLS Record Protocol provides connection security that has two basic properties:

- The connection is private. Symmetric cryptography is used for data encryption (e.g., DES [DES], RC4 [SCH], etc.). The keys for this symmetric encryption are generated uniquely for each connection and are based on a secret negotiated by another protocol (such as the TLS Handshake Protocol). The Record Protocol can also be used without encryption.

- The connection is reliable. Message transport includes a message integrity check using a keyed MAC. Secure hash functions (e.g., SHA, MD5, etc.) are used for MAC computations. The Record Protocol can operate without a MAC, but is generally only used in this mode while another protocol is using the Record Protocol as a transport for negotiating security parameters.

The TLS Record Protocol is used for encapsulation of various higher level protocols. One such encapsulated protocol, the TLS Handshake Protocol, allows the server and client to authenticate each other and to negotiate an encryption algorithm and cryptographic keys before the application protocol transmits or receives its first byte of data. The TLS Handshake Protocol provides connection security that has three basic properties:

- The peer's identity can be authenticated using asymmetric, or public key, cryptography (e.g., RSA [RSA], DSS [DSS], etc.). This authentication can be made optional, but is generally required for at least one of the peers.

- The negotiation of a shared secret is secure: the negotiated

secret is unavailable to eavesdroppers, and for any authenticated connection the secret cannot be obtained, even by an attacker who can place himself in the middle of the connection.

- The negotiation is reliable: no attacker can modify the negotiation communication without being detected by the parties to the communication.

One advantage of TLS is that it is application protocol independent. Higher level protocols can layer on top of the TLS Protocol transparently. The TLS standard, however, does not specify how protocols add security with TLS; the decisions on how to initiate TLS handshaking and how to interpret the authentication certificates exchanged are left up to the judgment of the designers and implementors of protocols which run on top of TLS.

1.1 Differences from TLS 1.1

This document is a revision of the TLS 1.1 [TLS1.1] protocol which contains improved flexibility, particularly for negotiation of cryptographic algorithms. The major changes are:

- Merged in TLS Extensions and AES Cipher Suites from external documents.

- Replacement of MD5/SHA-1 combination in the PRF

- Replacement of MD5/SHA-1 combination in the digitally-signed element.

- Allow the client to indicate which hash functions it supports.

- Allow the server to indicate which has functions it supports

1.1 Requirements Terminology

Keywords "MUST", "MUST NOT", "REQUIRED", "SHOULD", "SHOULD NOT" and "MAY" that appear in this document are to be interpreted as described in RFC 2119 [REQ].

2. Goals

The goals of TLS Protocol, in order of their priority, are:

1. Cryptographic security: TLS should be used to establish a secure connection between two parties.

2. Interoperability: Independent programmers should be able to develop applications utilizing TLS that will then be able to

successfully exchange cryptographic parameters without knowledge of one another's code.

3. Extensibility: TLS seeks to provide a framework into which new public key and bulk encryption methods can be incorporated as necessary. This will also accomplish two sub-goals: to prevent the need to create a new protocol (and risking the introduction of possible new weaknesses) and to avoid the need to implement an entire new security library.

4. Relative efficiency: Cryptographic operations tend to be highly CPU intensive, particularly public key operations. For this reason, the TLS protocol has incorporated an optional session caching scheme to reduce the number of connections that need to be established from scratch. Additionally, care has been taken to reduce network activity.

3. Goals of this document

This document and the TLS protocol itself are based on the SSL 3.0 Protocol Specification as published by Netscape. The differences between this protocol and SSL 3.0 are not dramatic, but they are significant enough that the various versions of TLS and SSL 3.0 do not interoperate (although each protocol incorporates a mechanism by which an implementation can back down prior versions. This document is intended primarily for readers who will be implementing the protocol and those doing cryptographic analysis of it. The specification has been written with this in mind, and it is intended to reflect the needs of those two groups. For that reason, many of the algorithm-dependent data structures and rules are included in the body of the text (as opposed to in an appendix), providing easier access to them.

This document is not intended to supply any details of service definition nor interface definition, although it does cover select areas of policy as they are required for the maintenance of solid security.

4. Presentation language

This document deals with the formatting of data in an external representation. The following very basic and somewhat casually defined presentation syntax will be used. The syntax draws from several sources in its structure. Although it resembles the programming language "C" in its syntax and XDR [XDR] in both its syntax and intent, it would be risky to draw too many parallels. The purpose of this presentation language is to document TLS only, not to have general application beyond that particular goal.

4.1. Basic block size

The representation of all data items is explicitly specified. The basic data block size is one byte (i.e. 8 bits). Multiple byte data items are concatenations of bytes, from left to right, from top to bottom. From the bytestream a multi-byte item (a numeric in the example) is formed (using C notation) by:

```
    value = (byte[0] << 8*(n-1)) | (byte[1] << 8*(n-2)) |
            ... | byte[n-1];
```

This byte ordering for multi-byte values is the commonplace network byte order or big endian format.

4.2. Miscellaneous

Comments begin with "/*" and end with "*/".

Optional components are denoted by enclosing them in "[[ ]]" double brackets.

Single byte entities containing uninterpreted data are of type opaque.

4.3. Vectors

A vector (single dimensioned array) is a stream of homogeneous data elements. The size of the vector may be specified at documentation time or left unspecified until runtime. In either case the length declares the number of bytes, not the number of elements, in the vector. The syntax for specifying a new type T' that is a fixed length vector of type T is

```
    T T'[n];
```

Here T' occupies n bytes in the data stream, where n is a multiple of the size of T. The length of the vector is not included in the encoded stream.

In the following example, Datum is defined to be three consecutive bytes that the protocol does not interpret, while Data is three consecutive Datum, consuming a total of nine bytes.

```
    opaque Datum[3];      /* three uninterpreted bytes */
    Datum Data[9];        /* 3 consecutive 3 byte vectors */
```

Variable length vectors are defined by specifying a subrange of legal lengths, inclusively, using the notation <floor..ceiling>. When encoded, the actual length precedes the vector's contents in the byte stream. The length will be in the form of a number consuming as many bytes as required to hold the vector's specified maximum (ceiling) length. A variable length vector with an actual length field of zero is referred to as an empty vector.

```
    T T'<floor..ceiling>;
```

In the following example, mandatory is a vector that must contain between 300 and 400 bytes of type opaque. It can never be empty. The actual length field consumes two bytes, a uint16, sufficient to represent the value 400 (see Section 4.4). On the other hand, longer can represent up to 800 bytes of data, or 400 uint16 elements, and it may be empty. Its encoding will include a two byte actual length field prepended to the vector. The length of an encoded vector must be an even multiple of the length of a single element (for example, a 17 byte vector of uint16 would be illegal).

```
    opaque mandatory<300..400>;
          /* length field is 2 bytes, cannot be empty */
    uint16 longer<0..800>;
          /* zero to 400 16-bit unsigned integers */
```

4.4. Numbers

The basic numeric data type is an unsigned byte (uint8). All larger numeric data types are formed from fixed length series of bytes concatenated as described in Section 4.1 and are also unsigned. The following numeric types are predefined.

```
    uint8 uint16[2];
    uint8 uint24[3];
    uint8 uint32[4];
    uint8 uint64[8];
```

All values, here and elsewhere in the specification, are stored in "network" or "big-endian" order; the uint32 represented by the hex bytes 01 02 03 04 is equivalent to the decimal value 16909060.

4.5. Enumerateds

An additional sparse data type is available called enum. A field of type enum can only assume the values declared in the definition. Each definition is a different type. Only enumerateds of the same type may be assigned or compared. Every element of an enumerated must

be assigned a value, as demonstrated in the following example. Since the elements of the enumerated are not ordered, they can be assigned any unique value, in any order.

```
    enum { e1(v1), e2(v2), ... , en(vn) [[, (n)]] } Te;
```

Enumerateds occupy as much space in the byte stream as would its maximal defined ordinal value. The following definition would cause one byte to be used to carry fields of type Color.

```
    enum { red(3), blue(5), white(7) } Color;
```

One may optionally specify a value without its associated tag to force the width definition without defining a superfluous element. In the following example, Taste will consume two bytes in the data stream but can only assume the values 1, 2 or 4.

```
    enum { sweet(1), sour(2), bitter(4), (32000) } Taste;
```

The names of the elements of an enumeration are scoped within the defined type. In the first example, a fully qualified reference to the second element of the enumeration would be Color.blue. Such qualification is not required if the target of the assignment is well specified.

```
    Color color = Color.blue;     /* overspecified, legal */
    Color color = blue;           /* correct, type implicit */
```

For enumerateds that are never converted to external representation, the numerical information may be omitted.

```
    enum { low, medium, high } Amount;
```

4.6. Constructed types

Structure types may be constructed from primitive types for convenience. Each specification declares a new, unique type. The syntax for definition is much like that of C.

```
    struct {
      T1 f1;
      T2 f2;
      ...
      Tn fn;
    } [[T]];
```

The fields within a structure may be qualified using the type's name using a syntax much like that available for enumerateds. For example, T.f2 refers to the second field of the previous declaration. Structure definitions may be embedded.

4.6.1. Variants

Defined structures may have variants based on some knowledge that is available within the environment. The selector must be an enumerated type that defines the possible variants the structure defines. There must be a case arm for every element of the enumeration declared in the select. The body of the variant structure may be given a label for reference. The mechanism by which the variant is selected at runtime is not prescribed by the presentation language.

```
    struct {
        T1 f1;
        T2 f2;
        ....
        Tn fn;
        select (E) {
            case e1: Te1;
            case e2: Te2;
            ....
            case en: Ten;
        } [[fv]];
    } [[Tv]];
```

For example:

```
    enum { apple, orange } VariantTag;
    struct {
        uint16 number;
        opaque string<0..10>; /* variable length */
    } V1;
    struct {
        uint32 number;
        opaque string[10];    /* fixed length */
    } V2;
    struct {
        select (VariantTag) { /* value of selector is implicit */
            case apple: V1;   /* VariantBody, tag = apple */
            case orange: V2;  /* VariantBody, tag = orange */
        } variant_body;       /* optional label on variant */
    } VariantRecord;
```

Variant structures may be qualified (narrowed) by specifying a value for the selector prior to the type. For example, a

```
    orange VariantRecord
```

is a narrowed type of a VariantRecord containing a variant_body of type V2.

4.7. Cryptographic attributes

The four cryptographic operations digital signing, stream cipher encryption, block cipher encryption, and public key encryption are designated digitally-signed, stream-ciphered, block-ciphered, and public-key-encrypted, respectively. A field's cryptographic processing is specified by prepending an appropriate key word designation before the field's type specification. Cryptographic keys are implied by the current session state (see Section 6.1).

In digital signing, one-way hash functions are used as input for a signing algorithm. A digitally-signed element is encoded as an opaque vector <0..2^16-1>, where the length is specified by the signing algorithm and key.

In RSA signing, the output of the chosen hash function is encoded as a PKCS #1 DigestInfo and then signed using block type 01 as described in Section 8.1 as described in [PKCS1A].

Note: the standard reference for PKCS#1 is now RFC 3447 [PKCS1B]. However, to minimize differences with TLS 1.0 text, we are using the terminology of RFC 2313 [PKCS1A].

In DSS, the 20 bytes of the SHA-1 hash are run directly through the Digital Signing Algorithm with no additional hashing. This produces two values, r and s. The DSS signature is an opaque vector, as above, the contents of which are the DER encoding of:

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

In stream cipher encryption, the plaintext is exclusive-ORed with an identical amount of output generated from a cryptographically-secure keyed pseudorandom number generator.

In block cipher encryption, every block of plaintext encrypts to a block of ciphertext. All block cipher encryption is done in CBC (Cipher Block Chaining) mode, and all items which are block-ciphered will be an exact multiple of the cipher block length.

In public key encryption, a public key algorithm is used to encrypt

data in such a way that it can be decrypted only with the matching private key. A public-key-encrypted element is encoded as an opaque vector <0..2^16-1>, where the length is specified by the signing algorithm and key.

An RSA encrypted value is encoded with PKCS #1 block type 2 as described in [PKCS1A].

In the following example:

```
    stream-ciphered struct {
        uint8 field1;
        uint8 field2;
        digitally-signed opaque hash[20];
    } UserType;
```

The contents of hash are used as input for the signing algorithm, then the entire structure is encrypted with a stream cipher. The length of this structure, in bytes would be equal to 2 bytes for field1 and field2, plus two bytes for the length of the signature, plus the length of the output of the signing algorithm. This is known due to the fact that the algorithm and key used for the signing are known prior to encoding or decoding this structure.

4.8. Constants

Typed constants can be defined for purposes of specification by declaring a symbol of the desired type and assigning values to it. Under-specified types (opaque, variable length vectors, and structures that contain opaque) cannot be assigned values. No fields of a multi-element structure or vector may be elided.

For example,

```
    struct {
        uint8 f1;
        uint8 f2;
    } Example1;

    Example1 ex1 = {1, 4};  /* assigns f1 = 1, f2 = 4 */
```

5. HMAC and the pseudorandom function

A number of operations in the TLS record and handshake layer required a keyed MAC; this is a secure digest of some data protected by a secret. Forging the MAC is infeasible without knowledge of the MAC secret. The construction we use for this operation is known as HMAC, described in [HMAC].

In addition, a construction is required to do expansion of secrets into blocks of data for the purposes of key generation or validation. This pseudo-random function (PRF) takes as input a secret, a seed, and an identifying label and produces an output of arbitrary length.

First, we define a data expansion function, P_hash(secret, data) which uses a single hash function to expand a secret and seed into an arbitrary quantity of output:

```
    P_hash(secret, seed) = HMAC_hash(secret, A(1) + seed) +
                           HMAC_hash(secret, A(2) + seed) +
                           HMAC_hash(secret, A(3) + seed) + ...
```

Where + indicates concatenation.

```
A() is defined as:
    A(0) = seed
    A(i) = HMAC_hash(secret, A(i-1))
```

P_hash can be iterated as many times as is necessary to produce the required quantity of data. For example, if P_SHA-1 was being used to create 64 bytes of data, it would have to be iterated 4 times (through A(4)), creating 80 bytes of output data; the last 16 bytes of the final iteration would then be discarded, leaving 64 bytes of output data.

TLS's PRF is created by applying P_hash to the secret S. The hash function used in P MUST be the same hash function selected for the HMAC in the cipher suite.

The label is an ASCII string. It should be included in the exact form it is given without a length byte or trailing null character. For example, the label "slithy toves" would be processed by hashing the following bytes:

```
    73 6C 69 74 68 79 20 74 6F 76 65 73
```

6. The TLS Record Protocol

The TLS Record Protocol is a layered protocol. At each layer, messages may include fields for length, description, and content. The Record Protocol takes messages to be transmitted, fragments the data into manageable blocks, optionally compresses the data, applies a MAC, encrypts, and transmits the result. Received data is decrypted, verified, decompressed, and reassembled, then delivered to higher level clients.

Four record protocol clients are described in this document: the handshake protocol, the alert protocol, the change cipher spec protocol, and the application data protocol. In order to allow extension of the TLS protocol, additional record types can be supported by the record protocol. Any new record types SHOULD allocate type values immediately beyond the ContentType values for the four record types described here (see Appendix A.1). All such values must be defined by RFC 2434 Standards Action. See section 11 for IANA Considerations for ContentType values.

If a TLS implementation receives a record type it does not understand, it SHOULD just ignore it. Any protocol designed for use over TLS MUST be carefully designed to deal with all possible attacks against it. Note that because the type and length of a record are not protected by encryption, care SHOULD be taken to minimize the value of traffic analysis of these values.

6.1. Connection states

A TLS connection state is the operating environment of the TLS Record Protocol. It specifies a compression algorithm, encryption algorithm, and MAC algorithm. In addition, the parameters for these algorithms are known: the MAC secret and the bulk encryption keys for the connection in both the read and the write directions. Logically, there are always four connection states outstanding: the current read and write states, and the pending read and write states. All records are processed under the current read and write states. The security parameters for the pending states can be set by the TLS Handshake Protocol, and the Change Cipher Spec can selectively make either of the pending states current, in which case the appropriate current state is disposed of and replaced with the pending state; the pending state is then reinitialized to an empty state. It is illegal to make a state which has not been initialized with security parameters a current state. The initial current state always specifies that no encryption, compression, or MAC will be used.

The security parameters for a TLS Connection read and write state are set by providing the following values:

connection end
    Whether this entity is considered the "client" or the "server" in this connection.

bulk encryption algorithm
    An algorithm to be used for bulk encryption. This specification includes the key size of this algorithm, how much of that key is secret, whether it is a block or stream cipher, the block size of the cipher (if appropriate).

MAC algorithm
    An algorithm to be used for message authentication. This specification includes the size of the hash which is returned by the MAC algorithm.

compression algorithm
    An algorithm to be used for data compression. This specification must include all information the algorithm requires to do

compression.

master secret
    A 48 byte secret shared between the two peers in the connection.

client random
    A 32 byte value provided by the client.

server random
    A 32 byte value provided by the server.

These parameters are defined in the presentation language as:

```
enum { server, client } ConnectionEnd;

enum { null, rc4, rc2, des, 3des, des40, idea, aes } BulkCipherAlgorithm

enum { stream, block } CipherType;

enum { null, md5, sha } MACAlgorithm;

enum { null(0), (255) } CompressionMethod;

/* The algorithms specified in CompressionMethod,
   BulkCipherAlgorithm, and MACAlgorithm may be added to. */

struct {
    ConnectionEnd          entity;
    BulkCipherAlgorithm    bulk_cipher_algorithm;
    CipherType             cipher_type;
    uint8                  key_size;
    uint8                  key_material_length;
    MACAlgorithm           mac_algorithm;
    uint8                  hash_size;
    CompressionMethod      compression_algorithm;
    opaque                 master_secret[48];
    opaque                 client_random[32];
    opaque                 server_random[32];
} SecurityParameters;
```

The record layer will use the security parameters to generate the following four items:

```
client write MAC secret
server write MAC secret
```

    client write key
    server write key

The client write parameters are used by the server when receiving and processing records and vice-versa. The algorithm used for generating these items from the security parameters is described in section 6.3.

Once the security parameters have been set and the keys have been generated, the connection states can be instantiated by making them the current states. These current states MUST be updated for each record processed. Each connection state includes the following elements:

compression state
    The current state of the compression algorithm.

cipher state
    The current state of the encryption algorithm. This will consist of the scheduled key for that connection. For stream ciphers, this will also contain whatever the necessary state information is to allow the stream to continue to encrypt or decrypt data.

MAC secret
    The MAC secret for this connection as generated above.

sequence number
    Each connection state contains a sequence number, which is maintained separately for read and write states. The sequence number MUST be set to zero whenever a connection state is made the active state. Sequence numbers are of type uint64 and may not exceed 2^64-1. Sequence numbers do not wrap. If a TLS implementation would need to wrap a sequence number it must renegotiate instead. A sequence number is incremented after each record: specifically, the first record which is transmitted under a particular connection state MUST use sequence number 0.

6.2. Record layer

The TLS Record Layer receives uninterpreted data from higher layers in non-empty blocks of arbitrary size.

6.2.1. Fragmentation

The record layer fragments information blocks into TLSPlaintext records carrying data in chunks of 2^14 bytes or less. Client message

boundaries are not preserved in the record layer (i.e., multiple
client messages of the same ContentType MAY be coalesced into a
single TLSPlaintext record, or a single message MAY be fragmented
across several records).

```
    struct {
        uint8 major, minor;
    } ProtocolVersion;

    enum {
        change_cipher_spec(20), alert(21), handshake(22),
        application_data(23), (255)
    } ContentType;

    struct {
        ContentType type;
        ProtocolVersion version;
        uint16 length;
        opaque fragment[TLSPlaintext.length];
    } TLSPlaintext;
```

type
    The higher level protocol used to process the enclosed fragment.

version
    The version of the protocol being employed. This document
    describes TLS Version 1.2, which uses the version { 3, 3 }. The
    version value 3.3 is historical, deriving from the use of 3.1 for
    TLS 1.0. (See Appendix A.1).

length
    The length (in bytes) of the following TLSPlaintext.fragment.
    The length should not exceed 2^14.

fragment
    The application data. This data is transparent and treated as an
    independent block to be dealt with by the higher level protocol
    specified by the type field.

Note: Data of different TLS Record layer content types MAY be
interleaved. Application data is generally of lower precedence
for transmission than other content types. However, records MUST
be delivered to the network in the same order as they are
protected by the record layer. Recipients MUST receive and
process interleaved application layer traffic during handshakes

subsequent to the first one on a connection.

6.2.2. Record compression and decompression

All records are compressed using the compression algorithm defined in the current session state. There is always an active compression algorithm; however, initially it is defined as CompressionMethod.null. The compression algorithm translates a TLSPlaintext structure into a TLSCompressed structure. Compression functions are initialized with default state information whenever a connection state is made active.

Compression must be lossless and may not increase the content length by more than 1024 bytes. If the decompression function encounters a TLSCompressed.fragment that would decompress to a length in excess of 2^14 bytes, it should report a fatal decompression failure error.

```
    struct {
        ContentType type;       /* same as TLSPlaintext.type */
        ProtocolVersion version;/* same as TLSPlaintext.version */
        uint16 length;
        opaque fragment[TLSCompressed.length];
    } TLSCompressed;
```

length
    The length (in bytes) of the following TLSCompressed.fragment.
    The length should not exceed 2^14 + 1024.

fragment
    The compressed form of TLSPlaintext.fragment.

Note: A CompressionMethod.null operation is an identity operation; no fields are altered.

Implementation note:
    Decompression functions are responsible for ensuring that messages cannot cause internal buffer overflows.

6.2.3. Record payload protection

The encryption and MAC functions translate a TLSCompressed structure into a TLSCiphertext. The decryption functions reverse the process. The MAC of the record also includes a sequence number so that missing, extra or repeated messages are detectable.

```
    struct {
        ContentType type;
```

```
            ProtocolVersion version;
            uint16 length;
            select (CipherSpec.cipher_type) {
                case stream: GenericStreamCipher;
                case block: GenericBlockCipher;
            } fragment;
        } TLSCiphertext;
```

type
    The type field is identical to TLSCompressed.type.

version
    The version field is identical to TLSCompressed.version.

length
    The length (in bytes) of the following TLSCiphertext.fragment.
    The length may not exceed 2^14 + 2048.

fragment
    The encrypted form of TLSCompressed.fragment, with the MAC.

6.2.3.1. Null or standard stream cipher

Stream ciphers (including BulkCipherAlgorithm.null - see Appendix A.6) convert TLSCompressed.fragment structures to and from stream TLSCiphertext.fragment structures.

```
    stream-ciphered struct {
        opaque content[TLSCompressed.length];
        opaque MAC[CipherSpec.hash_size];
    } GenericStreamCipher;
```

The MAC is generated as:

```
    HMAC_hash(MAC_write_secret, seq_num + TLSCompressed.type +
                  TLSCompressed.version + TLSCompressed.length +
                  TLSCompressed.fragment));
```

where "+" denotes concatenation.

seq_num
    The sequence number for this record.

hash
    The hashing algorithm specified by
    SecurityParameters.mac_algorithm.

Note that the MAC is computed before encryption. The stream cipher

encrypts the entire block, including the MAC. For stream ciphers that do not use a synchronization vector (such as RC4), the stream cipher state from the end of one record is simply used on the subsequent packet. If the CipherSuite is TLS_NULL_WITH_NULL_NULL, encryption consists of the identity operation (i.e., the data is not encrypted and the MAC size is zero implying that no MAC is used). TLSCiphertext.length is TLSCompressed.length plus CipherSpec.hash_size.

6.2.3.2. CBC block cipher

For block ciphers (such as RC2, DES, or AES), the encryption and MAC functions convert TLSCompressed.fragment structures to and from block TLSCiphertext.fragment structures.

```
    block-ciphered struct {
        opaque IV[CipherSpec.block_length];
        opaque content[TLSCompressed.length];
        opaque MAC[CipherSpec.hash_size];
        uint8 padding[GenericBlockCipher.padding_length];
        uint8 padding_length;
    } GenericBlockCipher;
```

The MAC is generated as described in Section 6.2.3.1.

IV

Unlike previous versions of SSL and TLS, TLS 1.1 uses an explicit IV in order to prevent the attacks described by [CBCATT]. We recommend the following equivalently strong procedures. For clarity we use the following notation.

```
IV -- the transmitted value of the IV field in the
    GenericBlockCipher structure.
CBC residue -- the last ciphertext block of the previous record
mask -- the actual value which the cipher XORs with the
    plaintext prior to encryption of the first cipher block
    of the record.
```

In prior versions of TLS, there was no IV field and the CBC residue and mask were one and the same. See Sections 6.1, 6.2.3.2 and 6.3, of [TLS1.0] for details of TLS 1.0 IV handling.

One of the following two algorithms SHOULD be used to generate the per-record IV:

```
      (1) Generate a cryptographically strong random string R of
          length CipherSpec.block_length. Place R
          in the IV field. Set the mask to R. Thus, the first
```

```
          cipher block will be encrypted as E(R XOR Data).

      (2) Generate a cryptographically strong random number R of
          length CipherSpec.block_length and prepend it to the plaintext
          prior to encryption. In
          this case either:

          (a)   The cipher may use a fixed mask such as zero.
          (b) The CBC residue from the previous record may be used
              as the mask. This preserves maximum code compatibility
           with TLS 1.0 and SSL 3. It also has the advantage that
           it does not require the ability to quickly reset the IV,
           which is known to be a   problem on some systems.

           In either 2(a) or 2(b) the data (R || data) is fed into the
           encryption process. The first cipher block (containing
           E(mask XOR R) is placed in the IV field. The first
           block of content contains E(IV XOR data)
```

The following alternative procedure MAY be used: However, it has not been demonstrated to be equivalently cryptographically strong to the above procedures. The sender prepends a fixed block F to the plaintext (or alternatively a block generated with a weak PRNG). He then encrypts as in (2) above, using the CBC residue from the previous block as the mask for the prepended block. Note that in this case the mask for the first record transmitted by the application (the Finished) MUST be generated using a cryptographically strong PRNG.

The decryption operation for all three alternatives is the same. The receiver decrypts the entire GenericBlockCipher structure and then discards the first cipher block, corresponding to the IV component.

padding

Padding that is added to force the length of the plaintext to be an integral multiple of the block cipher's block length. The padding MAY be any length up to 255 bytes long, as long as it results in the TLSCiphertext.length being an integral multiple of the block length. Lengths longer than necessary might be desirable to frustrate attacks on a protocol based on analysis of the lengths of exchanged messages. Each uint8 in the padding data vector MUST be filled with the padding length value. The receiver MUST check this padding and SHOULD use the bad_record_mac alert to indicate padding errors.

padding_length
The padding length MUST be such that the total size of the

GenericBlockCipher structure is a multiple of the cipher's block length. Legal values range from zero to 255, inclusive. This length specifies the length of the padding field exclusive of the padding_length field itself.

The encrypted data length (TLSCiphertext.length) is one more than the sum of TLSCompressed.length, CipherSpec.hash_size, and padding_length.

Example: If the block length is 8 bytes, the content length (TLSCompressed.length) is 61 bytes, and the MAC length is 20 bytes, the length before padding is 82 bytes (this does not include the IV, which may or may not be encrypted, as discussed above). Thus, the padding length modulo 8 must be equal to 6 in order to make the total length an even multiple of 8 bytes (the block length). The padding length can be 6, 14, 22, and so on, through 254. If the padding length were the minimum necessary, 6, the padding would be 6 bytes, each containing the value 6. Thus, the last 8 octets of the GenericBlockCipher before block encryption would be xx 06 06 06 06 06 06 06, where xx is the last octet of the MAC.

Note: With block ciphers in CBC mode (Cipher Block Chaining), it is critical that the entire plaintext of the record be known before any ciphertext is transmitted. Otherwise it is possible for the attacker to mount the attack described in [CBCATT].

Implementation Note: Canvel et. al. [CBCTIME] have demonstrated a timing attack on CBC padding based on the time required to compute the MAC. In order to defend against this attack, implementations MUST ensure that record processing time is essentially the same whether or not the padding is correct. In general, the best way to to do this is to compute the MAC even if the padding is incorrect, and only then reject the packet. For instance, if the pad appears to be incorrect the implementation might assume a zero-length pad and then compute the MAC. This leaves a small timing channel, since MAC performance depends to some extent on the size of the data fragment, but it is not believed to be large enough to be exploitable due to the large block size of existing MACs and the small size of the timing signal.

6.3. Key calculation

The Record Protocol requires an algorithm to generate keys, and MAC secrets from the security parameters provided by the handshake protocol.

The master secret is hashed into a sequence of secure bytes, which are assigned to the MAC secrets and keys required by the current connection state (see Appendix A.6). CipherSpecs require a client write MAC secret, a server write MAC secret, a client write key, and a server write key, which are generated from the master secret in that order. Unused values are empty.

When generating keys and MAC secrets, the master secret is used as an entropy source.

To generate the key material, compute

```
    key_block = PRF(SecurityParameters.master_secret,
                       "key expansion",
                       SecurityParameters.server_random +
                       SecurityParameters.client_random);
```

until enough output has been generated. Then the key_block is partitioned as follows:

```
    client_write_MAC_secret[SecurityParameters.hash_size]
    server_write_MAC_secret[SecurityParameters.hash_size]
    client_write_key[SecurityParameters.key_material_length]
    server_write_key[SecurityParameters.key_material_length]
```

Implementation note:
The currently defined which requires the most material is AES_256_CBC_SHA, defined in [TLSAES]. It requires 2 x 32 byte keys and 2 x 20 byte MAC secrets, for a total 104 bytes of key material.

7. The TLS Handshaking Protocols

TLS has three subprotocols which are used to allow peers to agree

      upon security parameters for the record layer, authenticate
      themselves, instantiate negotiated security parameters, and
      report error conditions to each other.

      The Handshake Protocol is responsible for negotiating a session,
      which consists of the following items:

      session identifier
        An arbitrary byte sequence chosen by the server to identify an
        active or resumable session state.

      peer certificate
        X509v3 [X509] certificate of the peer. This element of the

        state may be null.

      compression method
        The algorithm used to compress data prior to encryption.

      cipher spec
        Specifies the bulk data encryption algorithm (such as null,
        DES, etc.) and a MAC algorithm (such as MD5 or SHA). It also
        defines cryptographic attributes such as the hash_size. (See
        Appendix A.6 for formal definition)

      master secret
        48-byte secret shared between the client and server.

      is resumable
        A flag indicating whether the session can be used to initiate
        new connections.

   These items are then used to create security parameters for use by
   the Record Layer when protecting application data. Many connections
   can be instantiated using the same session through the resumption
   feature of the TLS Handshake Protocol.

7.1. Change cipher spec protocol

   The change cipher spec protocol exists to signal transitions in
   ciphering strategies. The protocol consists of a single message,
   which is encrypted and compressed under the current (not the pending)
   connection state. The message consists of a single byte of value 1.

```
      struct {
          enum { change_cipher_spec(1), (255) } type;
```

```
    } ChangeCipherSpec;
```

The change cipher spec message is sent by both the client and server to notify the receiving party that subsequent records will be protected under the newly negotiated CipherSpec and keys. Reception of this message causes the receiver to instruct the Record Layer to immediately copy the read pending state into the read current state. Immediately after sending this message, the sender MUST instruct the record layer to make the write pending state the write active state. (See section 6.1.) The change cipher spec message is sent during the handshake after the security parameters have been agreed upon, but before the verifying finished message is sent (see section 7.4.11

Note: if a rehandshake occurs while data is flowing on a connection, the communicating parties may continue to send data using the old CipherSpec. However, once the ChangeCipherSpec has been sent, the new

CipherSpec MUST be used. The first side to send the ChangeCipherSpec does not know that the other side has finished computing the new keying material (e.g. if it has to perform a time consuming public key operation). Thus, a small window of time during which the recipient must buffer the data MAY exist. In practice, with modern machines this interval is likely to be fairly short.

7.2. Alert protocol

One of the content types supported by the TLS Record layer is the alert type. Alert messages convey the severity of the message and a description of the alert. Alert messages with a level of fatal result in the immediate termination of the connection. In this case, other connections corresponding to the session may continue, but the session identifier MUST be invalidated, preventing the failed session from being used to establish new connections. Like other messages, alert messages are encrypted and compressed, as specified by the current connection state.

```
    enum { warning(1), fatal(2), (255) } AlertLevel;

    enum {
        close_notify(0),
        unexpected_message(10),
        bad_record_mac(20),
        decryption_failed(21),
        record_overflow(22),
        decompression_failure(30),
        handshake_failure(40),
        no_certificate_RESERVED (41),
        bad_certificate(42),
        unsupported_certificate(43),
        certificate_revoked(44),
        certificate_expired(45),
        certificate_unknown(46),
        illegal_parameter(47),
        unknown_ca(48),
        access_denied(49),
        decode_error(50),
        decrypt_error(51),
        export_restriction_RESERVED(60),
        protocol_version(70),
        insufficient_security(71),
        internal_error(80),
        user_canceled(90),
        no_renegotiation(100),
        unsupported_extension(110),           /* new */
        certificate_unobtainable(111),        /* new */
```

```
           unrecognized_name(112),               /* new */
           bad_certificate_status_response(113), /* new */
           bad_certificate_hash_value(114),      /* new */
           (255)
       } AlertDescription;

       struct {
           AlertLevel level;
           AlertDescription description;
       } Alert;
```

7.2.1. Closure alerts

The client and the server must share knowledge that the connection is ending in order to avoid a truncation attack. Either party may initiate the exchange of closing messages.

close_notify
This message notifies the recipient that the sender will not send any more messages on this connection. Note that as of TLS 1.1, failure to properly close a connection no longer requires that a session not be resumed. This is a change from TLS 1.0 to conform with widespread implementation practice.

Either party may initiate a close by sending a close_notify alert. Any data received after a closure alert is ignored.

Unless some other fatal alert has been transmitted, each party is required to send a close_notify alert before closing the write side of the connection. The other party MUST respond with a close_notify alert of its own and close down the connection immediately, discarding any pending writes. It is not required for the initiator of the close to wait for the responding close_notify alert before closing the read side of the connection.

If the application protocol using TLS provides that any data may be carried over the underlying transport after the TLS connection is closed, the TLS implementation must receive the responding close_notify alert before indicating to the application layer that the TLS connection has ended. If the application protocol will not transfer any additional data, but will only close the underlying transport connection, then the implementation MAY choose to close the transport without waiting for the responding close_notify. No part of this standard should be taken to dictate the manner in which a usage profile for TLS manages its data transport, including when connections are opened or closed.

Note: It is assumed that closing a connection reliably delivers

pending data before destroying the transport.

7.2.2. Error alerts

Error handling in the TLS Handshake protocol is very simple. When an error is detected, the detecting party sends a message to the other party. Upon transmission or receipt of an fatal alert message, both parties immediately close the connection. Servers and clients MUST forget any session-identifiers, keys, and secrets associated with a failed connection. Thus, any connection terminated with a fatal alert MUST NOT be resumed. The following error alerts are defined:

unexpected_message
    An inappropriate message was received. This alert is always fatal and should never be observed in communication between proper implementations.

bad_record_mac
    This alert is returned if a record is received with an incorrect MAC. This alert also MUST be returned if an alert is sent because a TLSCiphertext decrypted in an invalid way: either it wasn't an even multiple of the block length, or its padding values, when checked, weren't correct. This message is always fatal.

decryption_failed
    This alert MAY be returned if a TLSCiphertext decrypted in an
    invalid way: either it wasn't an even multiple of the block
    length, or its padding values, when checked, weren't correct.
    This message is always fatal.

    Note: Differentiating between bad_record_mac and
    decryption_failed alerts may permit certain attacks against CBC
    mode as used in TLS [CBCATT]. It is preferable to uniformly use
    the bad_record_mac alert to hide the specific type of the error.

record_overflow
    A TLSCiphertext record was received which had a length more than
    2^14+2048 bytes, or a record decrypted to a TLSCompressed record
    with more than 2^14+1024 bytes. This message is always fatal.

decompression_failure
    The decompression function received improper input (e.g. data
    that would expand to excessive length). This message is always
    fatal.

handshake_failure
    Reception of a handshake_failure alert message indicates that the

    sender was unable to negotiate an acceptable set of security
    parameters given the options available. This is a fatal error.

no_certificate_RESERVED
    This alert was used in SSLv3 but not in TLS. It should not be
    sent by compliant implementations.

bad_certificate
    A certificate was corrupt, contained signatures that did not
    verify correctly, etc.

unsupported_certificate
    A certificate was of an unsupported type.

certificate_revoked
    A certificate was revoked by its signer.

certificate_expired
    A certificate has expired or is not currently valid.

certificate_unknown
Some other (unspecified) issue arose in processing the certificate, rendering it unacceptable.

illegal_parameter
A field in the handshake was out of range or inconsistent with other fields. This is always fatal.

unknown_ca
A valid certificate chain or partial chain was received, but the certificate was not accepted because the CA certificate could not be located or couldn't be matched with a known, trusted CA. This message is always fatal.

access_denied
A valid certificate was received, but when access control was applied, the sender decided not to proceed with negotiation. This message is always fatal.

decode_error
A message could not be decoded because some field was out of the specified range or the length of the message was incorrect. This message is always fatal.

decrypt_error
A handshake cryptographic operation failed, including being unable to correctly verify a signature, decrypt a key exchange, or validate a finished message.

export_restriction_RESERVED
This alert was used in TLS 1.0 but not TLS 1.1.

protocol_version
The protocol version the client has attempted to negotiate is recognized, but not supported. (For example, old protocol versions might be avoided for security reasons). This message is always fatal.

insufficient_security
Returned instead of handshake_failure when a negotiation has failed specifically because the server requires ciphers more secure than those supported by the client. This message is always fatal.

internal_error
An internal error unrelated to the peer or the correctness of the

protocol makes it impossible to continue (such as a memory allocation failure). This message is always fatal.

user_canceled
This handshake is being canceled for some reason unrelated to a protocol failure. If the user cancels an operation after the handshake is complete, just closing the connection by sending a close_notify is more appropriate. This alert should be followed by a close_notify. This message is generally a warning.

no_renegotiation
Sent by the client in response to a hello request or by the server in response to a client hello after initial handshaking. Either of these would normally lead to renegotiation; when that is not appropriate, the recipient should respond with this alert; at that point, the original requester can decide whether to proceed with the connection. One case where this would be appropriate would be where a server has spawned a process to satisfy a request; the process might receive security parameters (key length, authentication, etc.) at startup and it might be difficult to communicate changes to these parameters after that point. This message is always a warning.

The following error alerts apply only to the extensions described in Section XXX. To avoid "breaking" existing clients and servers, these alerts MUST NOT be sent unless the sending party has received an extended hello message from the party they are communicating with.

unsupported_extension
sent by clients that receive an extended server hello containing

an extension that they did not put in the corresponding client hello (see Section 2.3). This message is always fatal.

unrecognized_name
sent by servers that receive a server_name extension request, but do not recognize the server name. This message MAY be fatal.

certificate_unobtainable
sent by servers who are unable to retrieve a certificate chain from the URL supplied by the client (see Section 3.3). This message MAY be fatal - for example if client authentication is required by the server for the handshake to continue and the server is unable to retrieve the certificate chain, it may send a fatal alert.

bad_certificate_status_response
    sent by clients that receive an invalid certificate status response (see Section 3.6). This message is always fatal.

bad_certificate_hash_value
    sent by servers when a certificate hash does not match a client provided certificate_hash. This message is always fatal.

For all errors where an alert level is not explicitly specified, the sending party MAY determine at its discretion whether this is a fatal error or not; if an alert with a level of warning is received, the receiving party MAY decide at its discretion whether to treat this as a fatal error or not. However, all messages which are transmitted with a level of fatal MUST be treated as fatal messages.

New alerts values MUST be defined by RFC 2434 Standards Action. See Section 11 for IANA Considerations for alert values.

7.3. Handshake Protocol overview

The cryptographic parameters of the session state are produced by the TLS Handshake Protocol, which operates on top of the TLS Record Layer. When a TLS client and server first start communicating, they agree on a protocol version, select cryptographic algorithms, optionally authenticate each other, and use public-key encryption techniques to generate shared secrets.

The TLS Handshake Protocol involves the following steps:

- Exchange hello messages to agree on algorithms, exchange random values, and check for session resumption.

- Exchange the necessary cryptographic parameters to allow the

client and server to agree on a premaster secret.

- Exchange certificates and cryptographic information to allow the client and server to authenticate themselves.

- Generate a master secret from the premaster secret and exchanged random values.

- Provide security parameters to the record layer.

- Allow the client and server to verify that their peer has

    calculated the same security parameters and that the handshake
    occurred without tampering by an attacker.

Note that higher layers should not be overly reliant on TLS always negotiating the strongest possible connection between two peers: there are a number of ways a man in the middle attacker can attempt to make two entities drop down to the least secure method they support. The protocol has been designed to minimize this risk, but there are still attacks available: for example, an attacker could block access to the port a secure service runs on, or attempt to get the peers to negotiate an unauthenticated connection. The fundamental rule is that higher levels must be cognizant of what their security requirements are and never transmit information over a channel less secure than what they require. The TLS protocol is secure, in that any cipher suite offers its promised level of security: if you negotiate 3DES with a 1024 bit RSA key exchange with a host whose certificate you have verified, you can expect to be that secure.

However, you SHOULD never send data over a link encrypted with 40 bit security unless you feel that data is worth no more than the effort required to break that encryption.

These goals are achieved by the handshake protocol, which can be summarized as follows: The client sends a client hello message to which the server must respond with a server hello message, or else a fatal error will occur and the connection will fail. The client hello and server hello are used to establish security enhancement capabilities between client and server. The client hello and server hello establish the following attributes: Protocol Version, Session ID, Cipher Suite, and Compression Method. Additionally, two random values are generated and exchanged: ClientHello.random and ServerHello.random.

The actual key exchange uses up to four messages: the server certificate, the server key exchange, the client certificate, and the client key exchange. New key exchange methods can be created by specifying a format for these messages and defining the use of the messages to allow the client and server to agree upon a shared secret. This secret MUST be quite long; currently defined key exchange methods exchange secrets which range from 48 to 128 bytes in length.

Following the hello messages, the server will send its certificate, if it is to be authenticated. Additionally, a server key exchange message may be sent, if it is required (e.g. if their server has no certificate, or if its certificate is for signing only). If the server is authenticated, it may request a certificate from the client, if that is appropriate to the cipher suite selected. Now the server will send the server hello done message, indicating that the hello-message phase of the handshake is complete. The server will then wait for a client response. If the server has sent a certificate request message, the client must send the certificate message. The client key exchange message is now sent, and the content of that message will depend on the public key algorithm selected between the client hello and the server hello. If the client has sent a certificate with signing ability, a digitally-signed certificate verify message is sent to explicitly verify the certificate.

At this point, a change cipher spec message is sent by the client, and the client copies the pending Cipher Spec into the current Cipher Spec. The client then immediately sends the finished message under the new algorithms, keys, and secrets. In response, the server will send its own change cipher spec message, transfer the pending to the current Cipher Spec, and send its finished message under the new

Cipher Spec. At this point, the handshake is complete and the client and server may begin to exchange application layer data. (See flow chart below.) Application data MUST NOT be sent prior to the completion of the first handshake (before a cipher suite other TLS_NULL_WITH_NULL_NULL is established).

```
      Client                                               Server

      ClientHello                  -------->
                                                      ServerHello
                                                     Certificate*
                                               CertificateStatus*
                                               ServerKeyExchange*
                                              CertificateRequest*
                                   <--------      ServerHelloDone
      Certificate*
      CertificateURL*
      ClientKeyExchange
      CertificateVerify*
      [ChangeCipherSpec]
      Finished                     -------->
                                               [ChangeCipherSpec]
                                   <--------             Finished
      Application Data             <------->     Application Data
```

Fig. 1 - Message flow for a full handshake

* Indicates optional or situation-dependent messages that are not always sent.

Note: To help avoid pipeline stalls, ChangeCipherSpec is an independent TLS Protocol content type, and is not actually a TLS handshake message.

When the client and server decide to resume a previous session or duplicate an existing session (instead of negotiating new security parameters) the message flow is as follows:

The client sends a ClientHello using the Session ID of the session to be resumed. The server then checks its session cache for a match. If a match is found, and the server is willing to re-establish the connection under the specified session state, it will send a ServerHello with the same Session ID value. At this point, both client and server MUST send change cipher spec messages and proceed directly to finished messages. Once the re-establishment is complete, the client and server MAY begin to exchange application layer data. (See flow chart below.) If a Session ID match is not found, the server generates a new session ID and the TLS client and server perform a full handshake.

```
      Client                                                Server
      ClientHello                   -------->
                                                       ServerHello
                                                [ChangeCipherSpec]
                                    <--------             Finished
      [ChangeCipherSpec]
      Finished                      -------->
      Application Data              <------->     Application Data

         Fig. 2 - Message flow for an abbreviated handshake
```

The contents and significance of each message will be presented in detail in the following sections.

7.4. Handshake protocol

The TLS Handshake Protocol is one of the defined higher level clients of the TLS Record Protocol. This protocol is used to negotiate the secure attributes of a session. Handshake messages are supplied to the TLS Record Layer, where they are encapsulated within one or more TLSPlaintext structures, which are processed and transmitted as specified by the current active session state.

```
       enum {
           hello_request(0), client_hello(1), server_hello(2),
           certificate(11), server_key_exchange (12),
           certificate_request(13), server_hello_done(14),
           certificate_verify(15), client_key_exchange(16),
           finished(20), certificate_url(21), certificate_status(22),
        (255)
       } HandshakeType;

       struct {
           HandshakeType msg_type;    /* handshake type */
           uint24 length;             /* bytes in message */
           select (HandshakeType) {
               case hello_request:       HelloRequest;
               case client_hello:        ClientHello;
               case server_hello:        ServerHello;
               case certificate:         Certificate;
               case server_key_exchange: ServerKeyExchange;
               case certificate_request: CertificateRequest;
               case server_hello_done:   ServerHelloDone;
               case certificate_verify:  CertificateVerify;
               case client_key_exchange: ClientKeyExchange;
               case finished:            Finished;
               case certificate_url:     CertificateURL;
```

```
              case certificate_status:  CertificateStatus;
          } body;
      } Handshake;
```

The handshake protocol messages are presented below in the order they MUST be sent; sending handshake messages in an unexpected order results in a fatal error. Unneeded handshake messages can be omitted, however. Note one exception to the ordering: the Certificate message is used twice in the handshake (from server to client, then from client to server), but described only in its first position. The one message which is not bound by these ordering rules is the Hello Request message, which can be sent at any time, but which should be ignored by the client if it arrives in the middle of a handshake.

New Handshake message type values MUST be defined via RFC 2434 Standards Action. See Section 11 for IANA Considerations for these values.

7.4.1. Hello messages

The hello phase messages are used to exchange security enhancement capabilities between the client and server. When a new session begins, the Record Layer's connection state encryption, hash, and compression algorithms are initialized to null. The current connection state is used for renegotiation messages.

7.4.1.1. Hello request

When this message will be sent:
    The hello request message MAY be sent by the server at any time.

Meaning of this message:
    Hello request is a simple notification that the client should begin the negotiation process anew by sending a client hello message when convenient. This message will be ignored by the client if the client is currently negotiating a session. This message may be ignored by the client if it does not wish to renegotiate a session, or the client may, if it wishes, respond with a no_renegotiation alert. Since handshake messages are intended to have transmission precedence over application data, it is expected that the negotiation will begin before no more than a few records are received from the client. If the server sends a hello request but does not receive a client hello in response, it may close the connection with a fatal alert.

After sending a hello request, servers SHOULD not repeat the request until the subsequent handshake negotiation is complete.

   Structure of this message:
       struct { } HelloRequest;

 Note: This message MUST NOT be included in the message hashes which are
       maintained throughout the handshake and used in the finished
       messages and the certificate verify message.

7.4.1.2. Client hello

   When this message will be sent:
       When a client first connects to a server it is required to send
       the client hello as its first message. The client can also send a
       client hello in response to a hello request or on its own
       initiative in order to renegotiate the security parameters in an
       existing connection.

       Structure of this message:
           The client hello message includes a random structure, which is
           used later in the protocol.

```
           struct {
              uint32 gmt_unix_time;
              opaque random_bytes[28];
           } Random;
```

       gmt_unix_time
       The current time and date in standard UNIX 32-bit format (seconds
       since the midnight starting Jan 1, 1970, GMT, ignoring leap
       seconds) according to the sender's internal clock. Clocks are not
       required to be set correctly by the basic TLS Protocol; higher
       level or application protocols may define additional
       requirements.

   random_bytes
       28 bytes generated by a secure random number generator.

   The client hello message includes a variable length session
   identifier. If not empty, the value identifies a session between the
   same client and server whose security parameters the client wishes to
   reuse. The session identifier MAY be from an earlier connection, this
   connection, or another currently active connection. The second option
   is useful if the client only wishes to update the random structures
   and derived values of a connection, while the third option makes it
   possible to establish several independent secure connections without
   repeating the full handshake protocol. These independent connections
   may occur sequentially or simultaneously; a SessionID becomes valid
   when the handshake negotiating it completes with the exchange of

Finished messages and persists until removed due to aging or because

a fatal error was encountered on a connection associated with the session. The actual contents of the SessionID are defined by the server.

```
    opaque SessionID<0..32>;
```

Warning:
    Because the SessionID is transmitted without encryption or immediate MAC protection, servers MUST not place confidential information in session identifiers or let the contents of fake session identifiers cause any breach of security. (Note that the content of the handshake as a whole, including the SessionID, is protected by the Finished messages exchanged at the end of the handshake.)

The CipherSuite list, passed from the client to the server in the client hello message, contains the combinations of cryptographic algorithms supported by the client in order of the client's preference (favorite choice first). Each CipherSuite defines a key exchange algorithm, a bulk encryption algorithm (including secret key length) and a MAC algorithm. The server will select a cipher suite or, if no acceptable choices are presented, return a handshake failure alert and close the connection.

```
    uint8 CipherSuite[2];    /* Cryptographic suite selector */
```

The client hello includes a list of compression algorithms supported by the client, ordered according to the client's preference.

```
    enum { null(0), (255) } CompressionMethod;

    struct {
        ProtocolVersion client_version;
        Random random;
        SessionID session_id;
        CipherSuite cipher_suites<2..2^16-1>;
        CompressionMethod compression_methods<1..2^8-1>;
    } ClientHello;
```

If the client wishes to use extensions (see Section XXX), it may send an ExtendedClientHello:

```
    struct {
        ProtocolVersion client_version;
        Random random;
        SessionID session_id;
        CipherSuite cipher_suites<2..2^16-1>;
        CompressionMethod compression_methods<1..2^8-1>;
```

```
        Extension client_hello_extension_list<0..2^16-1>;
    } ExtendedClientHello;
```

These two messages can be distinguished by determining whether there are bytes following what would be the end of the ClientHello.

client_version
    The version of the TLS protocol by which the client wishes to communicate during this session. This SHOULD be the latest (highest valued) version supported by the client. For this version of the specification, the version will be 3.2 (See Appendix E for details about backward compatibility).

random
    A client-generated random structure.

session_id
    The ID of a session the client wishes to use for this connection. This field should be empty if no session_id is available or the client wishes to generate new security parameters.

cipher_suites
    This is a list of the cryptographic options supported by the client, with the client's first preference first. If the session_id field is not empty (implying a session resumption request) this vector MUST include at least the cipher_suite from that session. Values are defined in Appendix A.5.

compression_methods
    This is a list of the compression methods supported by the client, sorted by client preference. If the session_id field is not empty (implying a session resumption request) it must include the compression_method from that session. This vector must contain, and all implementations must support, CompressionMethod.null. Thus, a client and server will always be able to agree on a compression method.

client_hello_extension_list
    Clients MAY request extended functionality from servers by sending data in the client_hello_extension_list. Here the new

"client_hello_extension_list" field contains a list of extensions. The actual "Extension" format is defined in Section XXX.

In the event that a client requests additional functionality using the extended client hello, and this functionality is not supplied by the server, the client MAY abort the handshake.

A server that supports the extensions mechanism MUST accept only client hello messages in either the original or extended ClientHello ormat, and (as for all other messages) MUST check that the amount of data in the message precisely matches one of these formats; if not then it MUST send a fatal "decode_error" alert.

After sending the client hello message, the client waits for a server hello message. Any other handshake message returned by the server except for a hello request is treated as a fatal error.

7.4.1.3. Server hello

When this message will be sent:
The server will send this message in response to a client hello message when it was able to find an acceptable set of algorithms. If it cannot find such a match, it will respond with a handshake failure alert.

Structure of this message:

```
struct {
    ProtocolVersion server_version;
    Random random;
    SessionID session_id;
    CipherSuite cipher_suite;
    CompressionMethod compression_method;
} ServerHello;
```

If the server is sending an extension, it should use the ExtendedServerHello:

```
    struct {
        ProtocolVersion server_version;
        Random random;
        SessionID session_id;
        CipherSuite cipher_suite;
```

```
       CompressionMethod compression_method;
    Extension server_hello_extension_list<0..2^16-1>;
   } ExtendedServerHello;
```

These two messages can be distinguished by determining whether there are bytes following what would be the end of the ServerHello.

server_version
This field will contain the lower of that suggested by the client in the client hello and the highest supported by the server. For this version of the specification, the version is 3.2 (See Appendix E for details about backward compatibility).

random
This structure is generated by the server and MUST be independently generated from the ClientHello.random.

session_id
This is the identity of the session corresponding to this connection. If the ClientHello.session_id was non-empty, the server will look in its session cache for a match. If a match is found and the server is willing to establish the new connection using the specified session state, the server will respond with the same value as was supplied by the client. This indicates a resumed session and dictates that the parties must proceed directly to the finished messages. Otherwise this field will contain a different value identifying the new session. The server may return an empty session_id to indicate that the session will not be cached and therefore cannot be resumed. If a session is resumed, it must be resumed using the same cipher suite it was originally negotiated with.

cipher_suite
The single cipher suite selected by the server from the list in ClientHello.cipher_suites. For resumed sessions this field is the value from the state of the session being resumed.

compression_method
The single compression algorithm selected by the server from the list in ClientHello.compression_methods. For resumed sessions this field is the value from the resumed session state.

server_hello_extension_list

A list of extensions. Note that only extensions offered by the client can appear in the server's list.

7.4.1.4 Hello Extensions

The extension format for extended client hellos and extended server hellos is:

```
       struct {
           ExtensionType extension_type;
           opaque extension_data<0..2^16-1>;
       } Extension;
```

Here:

  - "extension_type" identifies the particular extension type.

  - "extension_data" contains information specific to the particular extension type.

The extension types defined in this document are:

```
       enum {
           server_name(0), max_fragment_length(1),
           client_certificate_url(2), trusted_ca_keys(3),
           truncated_hmac(4), status_request(5),
        cert_hash_types(6), (65535)
       } ExtensionType;
```

The list of defined extension types is maintained by the IANA. The current list can be found at XXX (suggest http://www.iana.org/assignments/tls-extensions). See sections XXX and YYY for more information on how new values are added.

Note that for all extension types (including those defined in future), the extension type MUST NOT appear in the extended server hello unless the same extension type appeared in the corresponding client hello. Thus clients MUST abort the handshake if they receive an extension type in the extended server hello that they did not request in the associated (extended) client hello.

Nonetheless "server oriented" extensions may be provided in the future within this framework - such an extension, say of type x, would require the client to first send an extension of type x in the (extended) client hello with empty extension_data to indicate that it supports the extension type. In this case the client is offering the capability to understand the extension type, and the server is taking the client up on its offer.

Also note that when multiple extensions of different types are present in the extended client hello or the extended server hello, the extensions may appear in any order. There MUST NOT be more than one extension of the same type.

An extended client hello may be sent both when starting a new session and when requesting session resumption. Indeed a client that requests resumption of a session does not in general know whether the server will accept this request, and therefore it SHOULD send an extended client hello if it would normally do so for a new session. In general the specification of each extension type must include a

discussion of the effect of the extension both during new sessions and during resumed sessions.

Note also that all the extensions defined in this document are relevant only when a session is initiated. When a client includes one or more of the defined extension types in an extended client hello while requesting session resumption:

- If the resumption request is denied, the use of the extensions is negotiated as normal.

- If, on the other hand, the older session is resumed, then the server MUST ignore the extensions and send a server hello containing none of the extension types; in this case the functionality of these extensions negotiated during the original session initiation is applied to the resumed session.

7.4.1.4.1 Server Name Indication

[TLS1.1] does not provide a mechanism for a client to tell a server the name of the server it is contacting. It may be desirable for clients to provide this information to facilitate secure connections to servers that host multiple 'virtual' servers at a single underlying network address.

In order to provide the server name, clients MAY include an extension of type "server_name" in the (extended) client hello. The "extension_data" field of this extension SHALL contain "ServerNameList" where:

```
       struct {
           NameType name_type;
           select (name_type) {
               case host_name: HostName;
           } name;
       } ServerName;

       enum {
           host_name(0), (255)
       } NameType;

       opaque HostName<1..2^16-1>;

       struct {
           ServerName server_name_list<1..2^16-1>
       } ServerNameList;
```

Currently the only server names supported are DNS hostnames, however

this does not imply any dependency of TLS on DNS, and other name types may be added in the future (by an RFC that Updates this document). TLS MAY treat provided server names as opaque data and pass the names and types to the application.

"HostName" contains the fully qualified DNS hostname of the server, as understood by the client. The hostname is represented as a byte string using UTF-8 encoding [UTF8], without a trailing dot.

If the hostname labels contain only US-ASCII characters, then the client MUST ensure that labels are separated only by the byte 0x2E, representing the dot character U+002E (requirement 1 in section 3.1 of [IDNA] notwithstanding). If the server needs to match the HostName against names that contain non-US-ASCII characters, it MUST perform the conversion operation described in section 4 of [IDNA], treating the HostName as a "query string" (i.e. the AllowUnassigned flag MUST be set). Note that IDNA allows labels to be separated by any of the Unicode characters U+002E, U+3002, U+FF0E, and U+FF61, therefore servers MUST accept any of these characters as a label separator. If the server only needs to match the HostName against names containing exclusively ASCII characters, it MUST compare ASCII names case-insensitively.

Literal IPv4 and IPv6 addresses are not permitted in "HostName". It is RECOMMENDED that clients include an extension of type "server_name" in the client hello whenever they locate a server by a supported name type.

A server that receives a client hello containing the "server_name" extension, MAY use the information contained in the extension to guide its selection of an appropriate certificate to return to the client, and/or other aspects of security policy. In this event, the server SHALL include an extension of type "server_name" in the (extended) server hello. The "extension_data" field of this extension SHALL be empty.

If the server understood the client hello extension but does not recognize the server name, it SHOULD send an "unrecognized_name" alert (which MAY be fatal).

If an application negotiates a server name using an application protocol, then upgrades to TLS, and a server_name extension is sent, then the extension SHOULD contain the same name that was negotiated in the application protocol. If the server_name is established in the TLS session handshake, the client SHOULD NOT attempt to request a different server name at the application layer.

7.4.1.4.2 Maximum Fragment Length Negotiation

By default, TLS uses fixed maximum plaintext fragment length of 2^14 bytes. It may be desirable for constrained clients to negotiate a smaller maximum fragment length due to memory limitations or bandwidth limitations.

In order to negotiate smaller maximum fragment lengths, clients MAY include an extension of type "max_fragment_length" in the (extended) client hello. The "extension_data" field of this extension SHALL contain:

```
      enum{
          2^9(1), 2^10(2), 2^11(3), 2^12(4), (255)
      } MaxFragmentLength;
```

whose value is the desired maximum fragment length. The allowed values for this field are: 2^9, 2^10, 2^11, and 2^12.

Servers that receive an extended client hello containing a "max_fragment_length" extension, MAY accept the requested maximum fragment length by including an extension of type "max_fragment_length" in the (extended) server hello. The "extension_data" field of this extension SHALL contain "MaxFragmentLength" whose value is the same as the requested maximum fragment length.

If a server receives a maximum fragment length negotiation request for a value other than the allowed values, it MUST abort the handshake with an "illegal_parameter" alert. Similarly, if a client receives a maximum fragment length negotiation response that differs from the length it requested, it MUST also abort the handshake with an "illegal_parameter" alert.

Once a maximum fragment length other than 2^14 has been successfully negotiated, the client and server MUST immediately begin fragmenting messages (including handshake messages), to ensure that no fragment larger than the negotiated length is sent. Note that TLS already requires clients and servers to support fragmentation of handshake messages.

The negotiated length applies for the duration of the session including session resumptions.

The negotiated length limits the input that the record layer may process without fragmentation (that is, the maximum value of TLSPlaintext.length; see [TLS] section 6.2.1). Note that the output of the record layer may be larger. For example, if the negotiated length is 2^9=512, then for currently defined cipher suites and when null compression is used, the record layer output can be at most 793

bytes: 5 bytes of headers, 512 bytes of application data, 256 bytes of padding, and 20 bytes of MAC. That means that in this event a TLS record layer peer receiving a TLS record layer message larger than 793 bytes may discard the message and send a "record_overflow" alert, without decrypting the message.

7.4.1.4.3 Client Certificate URLs

Ordinarily, when client authentication is performed, client certificates are sent by clients to servers during the TLS handshake. It may be desirable for constrained clients to send certificate URLs in place of certificates, so that they do not need to store their certificates and can therefore save memory.

In order to negotiate to send certificate URLs to a server, clients MAY include an extension of type "client_certificate_url" in the (extended) client hello. The "extension_data" field of this extension SHALL be empty.

(Note that it is necessary to negotiate use of client certificate

URLs in order to avoid "breaking" existing TLS 1.0 servers.)

Servers that receive an extended client hello containing a "client_certificate_url" extension, MAY indicate that they are willing to accept certificate URLs by including an extension of type "client_certificate_url" in the (extended) server hello. The "extension_data" field of this extension SHALL be empty.

After negotiation of the use of client certificate URLs has been successfully completed (by exchanging hellos including "client_certificate_url" extensions), clients MAY send a "CertificateURL" message in place of a "Certificate" message. See Section XXX.

7.4.1.4.4 Trusted CA Indication

Constrained clients that, due to memory limitations, possess only a small number of CA root keys, may wish to indicate to servers which root keys they possess, in order to avoid repeated handshake failures.

In order to indicate which CA root keys they possess, clients MAY include an extension of type "trusted_ca_keys" in the (extended) client hello. The "extension_data" field of this extension SHALL contain "TrustedAuthorities" where:

```
        struct {
            TrustedAuthority trusted_authorities_list<0..2^16-1>;
```

```
        } TrustedAuthorities;

        struct {
            IdentifierType identifier_type;
            select (identifier_type) {
                case pre_agreed: struct {};
                case key_sha1_hash: SHA1Hash;
                case x509_name: DistinguishedName;
                case cert_sha1_hash: SHA1Hash;
            } identifier;
        } TrustedAuthority;

        enum {
            pre_agreed(0), key_sha1_hash(1), x509_name(2),
            cert_sha1_hash(3), (255)
        } IdentifierType;
```

```
      opaque DistinguishedName<1..2^16-1>;
```

Here "TrustedAuthorities" provides a list of CA root key identifiers that the client possesses. Each CA root key is identified via either:

- "pre_agreed" - no CA root key identity supplied.

- "key_sha1_hash" - contains the SHA-1 hash of the CA root key. For DSA and ECDSA keys, this is the hash of the "subjectPublicKey" value. For RSA keys, the hash is of the big-endian byte string representation of the modulus without any initial 0-valued bytes. (This copies the key hash formats deployed in other environments.)

- "x509_name" - contains the DER-encoded X.509 DistinguishedName of the CA.

- "cert_sha1_hash" - contains the SHA-1 hash of a DER-encoded Certificate containing the CA root key.

Note that clients may include none, some, or all of the CA root keys they possess in this extension.

Note also that it is possible that a key hash or a Distinguished Name alone may not uniquely identify a certificate issuer - for example if a particular CA has multiple key pairs - however here we assume this is the case following the use of Distinguished Names to identify certificate issuers in TLS.

The option to include no CA root keys is included to allow the client to indicate possession of some pre-defined set of CA root keys.

Servers that receive a client hello containing the "trusted_ca_keys" extension, MAY use the information contained in the extension to guide their selection of an appropriate certificate chain to return to the client. In this event, the server SHALL include an extension of type "trusted_ca_keys" in the (extended) server hello. The "extension_data" field of this extension SHALL be empty.

7.4.1.4.5 Truncated HMAC

Currently defined TLS cipher suites use the MAC construction HMAC with either MD5 or SHA-1 [HMAC] to authenticate record layer communications. In TLS the entire output of the hash function is used as the MAC tag. However it may be desirable in constrained environments to save bandwidth by truncating the output of the hash function to 80 bits when forming MAC tags.

In order to negotiate the use of 80-bit truncated HMAC, clients MAY include an extension of type "truncated_hmac" in the extended client hello. The "extension_data" field of this extension SHALL be empty.

Servers that receive an extended hello containing a "truncated_hmac" extension, MAY agree to use a truncated HMAC by including an extension of type "truncated_hmac", with empty "extension_data", in the extended server hello.

Note that if new cipher suites are added that do not use HMAC, and the session negotiates one of these cipher suites, this extension will have no effect. It is strongly recommended that any new cipher suites using other MACs consider the MAC size as an integral part of the cipher suite definition, taking into account both security and bandwidth considerations.

If HMAC truncation has been successfully negotiated during a TLS handshake, and the negotiated cipher suite uses HMAC, both the client and the server pass this fact to the TLS record layer along with the other negotiated security parameters. Subsequently during the session, clients and servers MUST use truncated HMACs, calculated as specified in [HMAC]. That is, CipherSpec.hash_size is 10 bytes, and only the first 10 bytes of the HMAC output are transmitted and checked. Note that this extension does not affect the calculation of the PRF as part of handshaking or key derivation.

The negotiated HMAC truncation size applies for the duration of the session including session resumptions.

7.4.1.4.6 Certificate Status Request

Constrained clients may wish to use a certificate-status protocol such as OCSP [OCSP] to check the validity of server certificates, in order to avoid transmission of CRLs and therefore save bandwidth on constrained networks. This extension allows for such information to be sent in the TLS handshake, saving roundtrips and resources.

In order to indicate their desire to receive certificate status information, clients MAY include an extension of type "status_request" in the (extended) client hello. The

"extension_data" field of this extension SHALL contain
"CertificateStatusRequest" where:

```
      struct {
          CertificateStatusType status_type;
          select (status_type) {
              case ocsp: OCSPStatusRequest;
          } request;
      } CertificateStatusRequest;

      enum { ocsp(1), (255) } CertificateStatusType;

      struct {
          ResponderID responder_id_list<0..2^16-1>;
          Extensions  request_extensions;
      } OCSPStatusRequest;

      opaque ResponderID<1..2^16-1>;
```

In the OCSPStatusRequest, the "ResponderIDs" provides a list of OCSP responders that the client trusts. A zero-length "responder_id_list" sequence has the special meaning that the responders are implicitly known to the server - e.g., by prior arrangement. "Extensions" is a DER encoding of OCSP request extensions.

Both "ResponderID" and "Extensions" are DER-encoded ASN.1 types as defined in [OCSP]. "Extensions" is imported from [PKIX]. A zero-length "request_extensions" value means that there are no extensions (as opposed to a zero-length ASN.1 SEQUENCE, which is not valid for the "Extensions" type).

In the case of the "id-pkix-ocsp-nonce" OCSP extension, [OCSP] is unclear about its encoding; for clarification, the nonce MUST be a DER-encoded OCTET STRING, which is encapsulated as another OCTET STRING (note that implementations based on an existing OCSP client will need to be checked for conformance to this requirement).

Servers that receive a client hello containing the "status_request" extension, MAY return a suitable certificate status response to the client along with their certificate. If OCSP is requested, they SHOULD use the information contained in the extension when selecting an OCSP responder, and SHOULD include request_extensions in the OCSP request.

Servers return a certificate response along with their certificate by sending a "CertificateStatus" message immediately after the "Certificate" message (and before any "ServerKeyExchange" or "CertificateRequest" messages). Section XXX describes the CertificateStatus message.

7.4.1.4.7 Cert Hash Types

The client MAY use the "cert_hash_types" to indicate to the server which hash functions may be used in the signature on the server's certificate. The "extension_data" field of this extension contains:

```
     enum{
         md5(0), sha1(1), sha256(2), sha512(3), (255)
     } HashType;

     struct {
           HashType<255> types;
     } CertHashTypes;
```

These values indicate support for MD5 [MD5], SHA-1, SHA-256, and SHA-512 [SHA] respectively. The server MUST NOT send this extension.

Clients SHOULD send this extension if they support any algorithm other than SHA-1. If this extension is not used, servers SHOULD assume that the client supports only SHA-1. Note: this is a change from TLS 1.1 where there are no explicit rules but as a practical matter one can assume that the peer supports MD5 and SHA-1.

HashType values are divided into three groups:

1. Values from 0 (zero) through 63 decimal (0x3F) inclusive are reserved for IETF Standards Track protocols.

2. Values from 64 decimal (0x40) through 223 decimal (0xDF) inclusive are reserved for assignment for non-Standards Track methods.

3. Values from 224 decimal (0xE0) through 255 decimal (0xFF) inclusive are reserved for private use.

Additional information describing the role of IANA in the

allocation of HashType code points is described in Section 11.

7.4.1.4.8 Procedure for Defining New Extensions

The list of extension types, as defined in Section 2.3, is maintained by the Internet Assigned Numbers Authority (IANA). Thus an application needs to be made to the IANA in order to obtain a new extension type value. Since there are subtle (and not so subtle) interactions that may occur in this protocol between new features and existing features which may result in a significant reduction in overall security, new values SHALL be defined only through the IETF Consensus process specified in [IANA].

(This means that new assignments can be made only via RFCs approved by the IESG.)

The following considerations should be taken into account when designing new extensions:

- All of the extensions defined in this document follow the convention that for each extension that a client requests and that the server understands, the server replies with an extension of the same type.

- Some cases where a server does not agree to an extension are error conditions, and some simply a refusal to support a particular feature. In general error alerts should be used for the former, and a field in the server extension response for the latter.

- Extensions should as far as possible be designed to prevent any attack that forces use (or non-use) of a particular feature by manipulation of handshake messages. This principle should be followed regardless of whether the feature is believed to cause a security problem.

  Often the fact that the extension fields are included in the inputs to the Finished message hashes will be sufficient, but extreme care is needed when the extension changes the meaning of messages sent in the handshake phase. Designers and implementors should be aware of the fact that until the handshake has been authenticated, active attackers can modify messages and insert, remove, or replace extensions.

- It would be technically possible to use extensions to change major aspects of the design of TLS; for example the design of cipher suite negotiation. This is not recommended; it would be more

appropriate to define a new version of TLS - particularly since the TLS handshake algorithms have specific protection against version rollback attacks based on the version number, and the possibility of version rollback should be a significant consideration in any major design change.

7.4.2. Server certificate

When this message will be sent:
The server MUST send a certificate whenever the agreed-upon key exchange method is not an anonymous one. This message will always immediately follow the server hello message.

Meaning of this message:
The certificate type MUST be appropriate for the selected cipher suite's key exchange algorithm, and is generally an X.509v3 certificate. It MUST contain a key which matches the key exchange method, as follows. Unless otherwise specified, the signing
algorithm for the certificate MUST be the same as the algorithm for the certificate key. Unless otherwise specified, the public key MAY be of any length.

Key Exchange Algorithm	Certificate Key Type
RSA	RSA public key; the certificate MUST allow the key to be used for encryption.
DHE_DSS	DSS public key.
DHE_RSA	RSA public key which can be used for signing.
DH_DSS	Diffie-Hellman key. The algorithm used to sign the certificate MUST be DSS.
DH_RSA	Diffie-Hellman key. The algorithm used to sign the certificate MUST be RSA.

All certificate profiles, key and cryptographic formats are defined by the IETF PKIX working group [PKIX]. When a key usage extension is present, the digitalSignature bit MUST be set for the key to be eligible for signing, as described above, and the keyEncipherment bit MUST be present to allow encryption, as described above. The keyAgreement bit must be set on Diffie-Hellman certificates.

As CipherSuites which specify new key exchange methods are specified

for the TLS Protocol, they will imply certificate format and the
required encoded keying information.

Structure of this message:

```
    opaque ASN.1Cert<1..2^24-1>;

    struct {
        ASN.1Cert certificate_list<0..2^24-1>;
    } Certificate;
```

certificate_list
    This is a sequence (chain) of X.509v3 certificates. The sender's certificate must come first in the list. Each following certificate must directly certify the one preceding it. Because certificate validation requires that root keys be distributed independently, the self-signed certificate which specifies the root certificate authority may optionally be omitted from the chain, under the assumption that the remote end must already possess it in order to validate it in any case.

The same message type and structure will be used for the client's response to a certificate request message. Note that a client MAY send no certificates if it does not have an appropriate certificate to send in response to the server's authentication request.

Note: PKCS #7 [PKCS7] is not used as the format for the certificate vector because PKCS #6 [PKCS6] extended certificates are not used. Also PKCS #7 defines a SET rather than a SEQUENCE, making the task of parsing the list more difficult.

7.4.3. Server key exchange message

When this message will be sent:
    This message will be sent immediately after the server certificate message (or the server hello message, if this is an anonymous negotiation).

    The server key exchange message is sent by the server only when the server certificate message (if sent) does not contain enough data to allow the client to exchange a premaster secret. This is true for the following key exchange methods:

        DHE_DSS
        DHE_RSA
        DH_anon

    It is not legal to send the server key exchange message for the following key exchange methods:

```
          RSA
          DH_DSS
          DH_RSA
```

Meaning of this message:
    This message conveys cryptographic information to allow the client to communicate the premaster secret: either an RSA public key to encrypt the premaster secret with, or a Diffie-Hellman public key with which the client can complete a key exchange (with the result being the premaster secret.)

As additional CipherSuites are defined for TLS which include new key exchange algorithms, the server key exchange message will be sent if and only if the certificate type associated with the key exchange algorithm does not provide enough information for the client to exchange a premaster secret.

If the SignatureAlgorithm being used to sign the ServerKeyExchange message is DSA, the hash function used MUST be SHA-1. If the SignatureAlgorithm it must be the same hash function used in the signature of the server's certificate (found in the Certificate) message. This algorithm is denoted Hash below. Hash.length is the length of the output of that algorithm.

Structure of this message:

```
    enum { rsa, diffie_hellman } KeyExchangeAlgorithm;

    struct {
        opaque rsa_modulus<1..2^16-1>;
        opaque rsa_exponent<1..2^16-1>;
    } ServerRSAParams;

    rsa_modulus
        The modulus of the server's temporary RSA key.

    rsa_exponent
        The public exponent of the server's temporary RSA key.

    struct {
        opaque dh_p<1..2^16-1>;
        opaque dh_g<1..2^16-1>;
        opaque dh_Ys<1..2^16-1>;
    } ServerDHParams;     /* Ephemeral DH parameters */

    dh_p
        The prime modulus used for the Diffie-Hellman operation.
```

```
       dh_g
```

```
           The generator used for the Diffie-Hellman operation.

       dh_Ys
           The server's Diffie-Hellman public value (g^X mod p).

       struct {
           select (KeyExchangeAlgorithm) {
               case diffie_hellman:
                   ServerDHParams params;
                   Signature signed_params;
               case rsa:
                   ServerRSAParams params;
                   Signature signed_params;
           };
       } ServerKeyExchange;

       struct {
           select (KeyExchangeAlgorithm) {
               case diffie_hellman:
                   ServerDHParams params;
               case rsa:
                   ServerRSAParams params;
           };
        } ServerParams;

       params
           The server's key exchange parameters.

       signed_params
           For non-anonymous key exchanges, a hash of the corresponding
           params value, with the signature appropriate to that hash
           applied.

       hash
           Hash(ClientHello.random + ServerHello.random + ServerParams)

       sha_hash
           SHA1(ClientHello.random + ServerHello.random + ServerParams)

       enum { anonymous, rsa, dsa } SignatureAlgorithm;


       struct {
           select (SignatureAlgorithm) {
               case anonymous: struct { };
               case rsa:
                   digitally-signed struct {
                 opaque hash[Hash.length];
```

```
                    };
                case dsa:
                    digitally-signed struct {
                        opaque sha_hash[20];
                    };
                };
            };
        } Signature;
```

7.4.4. CertificateStatus

If a server returns a "CertificateStatus" message, then the server MUST have included an extension of type "status_request" with empty "extension_data" in the extended server hello.

```
      struct {
          CertificateStatusType status_type;
          select (status_type) {
              case ocsp: OCSPResponse;
          } response;
      } CertificateStatus;

      opaque OCSPResponse<1..2^24-1>;
```

An "ocsp_response" contains a complete, DER-encoded OCSP response (using the ASN.1 type OCSPResponse defined in [OCSP]). Note that only one OCSP response may be sent.

The "CertificateStatus" message is conveyed using the handshake message type "certificate_status".

Note that a server MAY also choose not to send a "CertificateStatus" message, even if it receives a "status_request" extension in the client hello message.

Note in addition that servers MUST NOT send the "CertificateStatus" message unless it received a "status_request" extension in the client hello message.

Clients requesting an OCSP response, and receiving an OCSP response

in a "CertificateStatus" message MUST check the OCSP response and
abort the handshake if the response is not satisfactory.

7.4.5. Certificate request

When this message will be sent:

A non-anonymous server can optionally request a certificate from
the client, if appropriate for the selected cipher suite. This
message, if sent, will immediately follow the Server Key Exchange
message (if it is sent; otherwise, the Server Certificate
message).

Structure of this message:

```
    enum {
        rsa_sign(1), dss_sign(2), rsa_fixed_dh(3), dss_fixed_dh(4),
     rsa_ephemeral_dh_RESERVED(5), dss_ephemeral_dh_RESERVED(6),
     fortezza_dms_RESERVED(20),
        (255)
    } ClientCertificateType;


    opaque DistinguishedName<1..2^16-1>;

    struct {
        ClientCertificateType certificate_types<1..2^8-1>;
     HashType certificate_hash<1..2^8-1>;
        DistinguishedName certificate_authorities<0..2^16-1>;
    } CertificateRequest;
```

certificate_types
    This field is a list of the types of certificates requested,
    sorted in order of the server's preference.

certificate_types
    A list of the types of certificate types which the client may
    offer.

```
       rsa_sign        a certificate containing an RSA key
       dss_sign        a certificate containing a DSS key
       rsa_fixed_dh    a certificate signed with RSA and containing
                 a static DH key.
       dss_fixed_dh    a certificate signed with DSS and containing
                 a static DH key
```

Certificate types rsa_sign and dss_sign SHOULD contain certificates signed with the same algorithm. However, this is not required. This is a holdover from TLS 1.0 and 1.1.

certificate_hash
A list of acceptable hash algorithms to be used in certificate signatures.

certificate_authorities
A list of the distinguished names of acceptable certificate

authorities. These distinguished names may specify a desired distinguished name for a root CA or for a subordinate CA; thus, this message can be used both to describe known roots and a desired authorization space. If the certificate_authorities list is empty then the client MAY send any certificate of the appropriate ClientCertificateType, unless there is some external arrangement to the contrary.

ClientCertificateType values are divided into three groups:

1. Values from 0 (zero) through 63 decimal (0x3F) inclusive are reserved for IETF Standards Track protocols.

2. Values from 64 decimal (0x40) through 223 decimal (0xDF) inclusive are reserved for assignment for non-Standards Track methods.

3. Values from 224 decimal (0xE0) through 255 decimal (0xFF) inclusive are reserved for private use.

Additional information describing the role of IANA in the allocation of ClientCertificateType code points is described in Section 11.

Note: Values listed as RESERVED may not be used. They were used in SSLv3.

Note: DistinguishedName is derived from [X501]. DistinguishedNames are represented in DER-encoded format.

Note: It is a fatal handshake_failure alert for an anonymous server to request client authentication.

7.4.6. Server hello done

When this message will be sent:
The server hello done message is sent by the server to indicate the end of the server hello and associated messages. After sending this message the server will wait for a client response.

Meaning of this message:
This message means that the server is done sending messages to support the key exchange, and the client can proceed with its phase of the key exchange.

Upon receipt of the server hello done message the client SHOULD verify that the server provided a valid certificate if required and check that the server hello parameters are acceptable.

Structure of this message:

```
struct { } ServerHelloDone;
```

7.4.7. Client certificate

When this message will be sent:
This is the first message the client can send after receiving a server hello done message. This message is only sent if the server requests a certificate. If no suitable certificate is available, the client SHOULD send a certificate message containing no certificates. That is, the certificate_list structure has a length of zero. If client authentication is required by the server for the handshake to continue, it may respond with a fatal handshake failure alert. Client certificates are sent using the Certificate structure defined in Section 7.4.2.

Note: When using a static Diffie-Hellman based key exchange method (DH_DSS or DH_RSA), if client authentication is requested, the Diffie-Hellman group and generator encoded in the client's certificate MUST match the server specified Diffie-Hellman parameters if the client's parameters are to be used for the key exchange.

7.4.8. Client Certificate URLs

After negotiation of the use of client certificate URLs has been

successfully completed (by exchanging hellos including
"client_certificate_url" extensions), clients MAY send a
"CertificateURL" message in place of a "Certificate" message.

```
    enum {
        individual_certs(0), pkipath(1), (255)
    } CertChainType;

    enum {
        false(0), true(1)
    } Boolean;

    struct {
        CertChainType type;
        URLAndOptionalHash url_and_hash_list<1..2^16-1>;
    } CertificateURL;
```

```
    struct {
        opaque url<1..2^16-1>;
        Boolean hash_present;
        select (hash_present) {
            case false: struct {};
            case true: SHA1Hash;
        } hash;
    } URLAndOptionalHash;

    opaque SHA1Hash[20];
```

Here "url_and_hash_list" contains a sequence of URLs and optional
hashes.

When X.509 certificates are used, there are two possibilities:

- if CertificateURL.type is "individual_certs", each URL refers to
  a single DER-encoded X.509v3 certificate, with the URL for the
  client's certificate first, or

- if CertificateURL.type is "pkipath", the list contains a single
  URL referring to a DER-encoded certificate chain, using the type
  PkiPath described in Section 8.

When any other certificate format is used, the specification that
describes use of that format in TLS should define the encoding format
of certificates or certificate chains, and any constraint on their
ordering.

The hash corresponding to each URL at the client's discretion is either not present or is the SHA-1 hash of the certificate or certificate chain (in the case of X.509 certificates, the DER-encoded certificate or the DER-encoded PkiPath).

Note that when a list of URLs for X.509 certificates is used, the ordering of URLs is the same as that used in the TLS Certificate message (see [TLS] Section 7.4.2), but opposite to the order in which certificates are encoded in PkiPath. In either case, the self-signed root certificate MAY be omitted from the chain, under the assumption that the server must already possess it in order to validate it.

Servers receiving "CertificateURL" SHALL attempt to retrieve the client's certificate chain from the URLs, and then process the certificate chain as usual. A cached copy of the content of any URL in the chain MAY be used, provided that a SHA-1 hash is present for that URL and it matches the hash of the cached copy.

Servers that support this extension MUST support the http: URL scheme

for certificate URLs, and MAY support other schemes. Use of other schemes than "http", "https", or "ftp" may create unexpected problems.

If the protocol used is HTTP, then the HTTP server can be configured to use the Cache-Control and Expires directives described in [HTTP] to specify whether and for how long certificates or certificate chains should be cached.

The TLS server is not required to follow HTTP redirects when retrieving the certificates or certificate chain. The URLs used in this extension SHOULD therefore be chosen not to depend on such redirects.

If the protocol used to retrieve certificates or certificate chains returns a MIME formatted response (as HTTP does), then the following MIME Content-Types SHALL be used: when a single X.509v3 certificate is returned, the Content-Type is "application/pkix-cert" [PKIOP], and when a chain of X.509v3 certificates is returned, the Content-Type is "application/pkix-pkipath" (see Section XXX).

If a SHA-1 hash is present for an URL, then the server MUST check that the SHA-1 hash of the contents of the object retrieved from that URL (after decoding any MIME Content-Transfer-Encoding) matches the given hash. If any retrieved object does not have the correct SHA-1 hash, the server MUST abort the handshake with a "bad_certificate_hash_value" alert.

Note that clients may choose to send either "Certificate" or "CertificateURL" after successfully negotiating the option to send certificate URLs. The option to send a certificate is included to provide flexibility to clients possessing multiple certificates.

If a server encounters an unreasonable delay in obtaining certificates in a given CertificateURL, it SHOULD time out and signal a "certificate_unobtainable" error alert.

7.4.9. Client key exchange message

When this message will be sent:
This message is always sent by the client. It MUST immediately follow the client certificate message, if it is sent. Otherwise it MUST be the first message sent by the client after it receives the server hello done message.

Meaning of this message:
With this message, the premaster secret is set, either though direct transmission of the RSA-encrypted secret, or by the transmission of

Diffie-Hellman parameters which will allow each side to agree upon the same premaster secret. When the key exchange method is DH_RSA or DH_DSS, client certification has been requested, and the client was able to respond with a certificate which contained a Diffie-Hellman public key whose parameters (group and generator) matched those specified by the server in its certificate, this message MUST not contain any data.

Structure of this message:
The choice of messages depends on which key exchange method has been selected. See Section 7.4.3 for the KeyExchangeAlgorithm definition.

```
struct {
    select (KeyExchangeAlgorithm) {
        case rsa: EncryptedPreMasterSecret;
        case diffie_hellman: ClientDiffieHellmanPublic;
    } exchange_keys;
} ClientKeyExchange;
```

7.4.9.1. RSA encrypted premaster secret message

Meaning of this message:
If RSA is being used for key agreement and authentication, the client

generates a 48-byte premaster secret, encrypts it using the public key from the server's certificate or the temporary RSA key provided in a server key exchange message, and sends the result in an encrypted premaster secret message. This structure is a variant of the client key exchange message, not a message in itself.

```
   Structure of this message:
   struct {
       ProtocolVersion client_version;
       opaque random[46];
   } PreMasterSecret;
```

client_version
: The latest (newest) version supported by the client. This is used to detect version roll-back attacks. Upon receiving the premaster secret, the server SHOULD check that this value matches the value transmitted by the client in the client hello message.

random
: 46 securely-generated random bytes.

```
      struct {
          public-key-encrypted PreMasterSecret pre_master_secret;
      } EncryptedPreMasterSecret;
```

pre_master_secret
: This random value is generated by the client and is used to generate the master secret, as specified in Section 8.1.

Note: An attack discovered by Daniel Bleichenbacher [BLEI] can be used to attack a TLS server which is using PKCS#1 v 1.5 encoded RSA. The attack takes advantage of the fact that by failing in different ways, a TLS server can be coerced into revealing whether a particular message, when decrypted, is properly PKCS#1 v1.5 formatted or not.

The best way to avoid vulnerability to this attack is to treat incorrectly formatted messages in a manner indistinguishable from correctly formatted RSA blocks. Thus, when it receives an incorrectly formatted RSA block, a server should generate a random 48-byte value and proceed using it as the premaster secret. Thus, the server will act identically whether the received RSA block is correctly encoded or not.

[PKCS1B] defines a newer version of PKCS#1 encoding that is more secure against the Bleichenbacher attack. However, for maximal compatibility with TLS 1.0, TLS 1.1 retains the original encoding. No variants of the Bleichenbacher attack are known to exist provided that the above recommendations are followed.

Implementation Note: public-key-encrypted data is represented as an opaque vector <0..2^16-1> (see section 4.7). Thus the RSA-encrypted PreMasterSecret in a ClientKeyExchange is preceded by two length bytes. These bytes are redundant in the case of RSA because the EncryptedPreMasterSecret is the only data in the ClientKeyExchange and its length can therefore be unambiguously determined. The SSLv3 specification was not clear about the encoding of public-key-encrypted data and therefore many SSLv3 implementations do not include the the length bytes, encoding the RSA encrypted data directly in the ClientKeyExchange message.

This specification requires correct encoding of the EncryptedPreMasterSecret complete with length bytes. The resulting PDU is incompatible with many SSLv3 implementations. Implementors upgrading from SSLv3 must modify their implementations to generate and accept the correct encoding. Implementors who wish to be compatible with both SSLv3 and TLS should make their implementation's behavior dependent on the protocol version.

Implementation Note: It is now known that remote timing-based attacks on SSL are possible, at least when the client and server are on the same LAN. Accordingly, implementations which use static RSA

keys SHOULD use RSA blinding or some other anti-timing technique, as described in [TIMING].

Note: The version number in the PreMasterSecret MUST be the version offered by the client in the ClientHello, not the version negotiated for the connection. This feature is designed to prevent rollback attacks. Unfortunately, many implementations use the negotiated version instead and therefore checking the version number may lead to failure to interoperate with such incorrect client implementations. Client implementations MUST and Server implementations MAY check the version number. In practice, since the TLS handshake MACs prevent downgrade and no good attacks are known on those MACs, ambiguity is not considered a serious security risk. Note that if servers choose to to check the version number, they should randomize the PreMasterSecret in case of error, rather than generate an alert, in order to avoid variants on the Bleichenbacher attack. [KPR03]

7.4.9.2. Client Diffie-Hellman public value

Meaning of this message:
  This structure conveys the client's Diffie-Hellman public value (Yc) if it was not already included in the client's certificate. The encoding used for Yc is determined by the enumerated PublicValueEncoding. This structure is a variant of the client key exchange message, not a message in itself.

Structure of this message:

```
    enum { implicit, explicit } PublicValueEncoding;

    implicit
        If the client certificate already contains a suitable Diffie-
        Hellman key, then Yc is implicit and does not need to be sent
        again. In this case, the client key exchange message will be
        sent, but MUST be empty.

    explicit
        Yc needs to be sent.

    struct {
        select (PublicValueEncoding) {
            case implicit: struct { };
            case explicit: opaque dh_Yc<1..2^16-1>;
        } dh_public;
    } ClientDiffieHellmanPublic;

    dh_Yc
        The client's Diffie-Hellman public value (Yc).
```

7.4.10. Certificate verify

When this message will be sent:
  This message is used to provide explicit verification of a client certificate. This message is only sent following a client certificate that has signing capability (i.e. all certificates except those containing fixed Diffie-Hellman parameters). When sent, it MUST immediately follow the client key exchange message.

Structure of this message:

```
    struct {
         Signature signature;
    } CertificateVerify;
```

The Signature type is defined in 7.4.3. If the SignatureAlgorithm is DSA, then the sha_hash value must be used. If it is RSA, the same function (denoted Hash) must be used as was used to create the signature for the client's certificate.

```
CertificateVerify.signature.hash
    Hash(handshake_messages);

CertificateVerify.signature.sha_hash
    SHA(handshake_messages);
```

Here handshake_messages refers to all handshake messages sent or received starting at client hello up to but not including this message, including the type and length fields of the handshake messages. This is the concatenation of all the Handshake structures as defined in 7.4 exchanged thus far.

7.4.10. Finished

When this message will be sent:
A finished message is always sent immediately after a change cipher spec message to verify that the key exchange and authentication processes were successful. It is essential that a change cipher spec message be received between the other handshake messages and the Finished message.

Meaning of this message:
The finished message is the first protected with the just-negotiated algorithms, keys, and secrets. Recipients of finished messages MUST verify that the contents are correct. Once a side has sent its Finished message and received and validated the Finished message from its peer, it may begin to send and receive application data over the connection.

```
struct {
    opaque verify_data[12];
} Finished;

verify_data
    PRF(master_secret, finished_label, MD5(handshake_messages) +
    SHA-1(handshake_messages)) [0..11];

finished_label
    For Finished messages sent by the client, the string "client
    finished". For Finished messages sent by the server, the
```

          string "server finished".

      handshake_messages
          All of the data from all messages in this handshake (not
          including any HelloRequest messages) up to but not including
          this message. This is only data visible at the handshake
          layer and does not include record layer headers. This is the
          concatenation of all the Handshake structures as defined in
          7.4 exchanged thus far.

It is a fatal error if a finished message is not preceded by a change cipher spec message at the appropriate point in the handshake.

The value handshake_messages includes all handshake messages starting at client hello up to, but not including, this finished message. This may be different from handshake_messages in Section 7.4.10 because it would include the certificate verify message (if sent). Also, the handshake_messages for the finished message sent by the client will be different from that for the finished message sent by the server, because the one which is sent second will include the prior one.

Note: Change cipher spec messages, alerts and any other record types are not handshake messages and are not included in the hash computations. Also, Hello Request messages are omitted from handshake hashes.

8. Cryptographic computations

In order to begin connection protection, the TLS Record Protocol requires specification of a suite of algorithms, a master secret, and the client and server random values. The authentication, encryption, and MAC algorithms are determined by the cipher_suite selected by the server and revealed in the server hello message. The compression algorithm is negotiated in the hello messages, and the random values are exchanged in the hello messages. All that remains is to calculate the master secret.

8.1. Computing the master secret

For all key exchange methods, the same algorithm is used to convert the pre_master_secret into the master_secret. The pre_master_secret should be deleted from memory once the master_secret has been computed.

```
       master_secret = PRF(pre_master_secret, "master secret",
                               ClientHello.random + ServerHello.random)
    [0..47];
```

The master secret is always exactly 48 bytes in length. The length of the premaster secret will vary depending on key exchange method.

8.1.1. RSA

When RSA is used for server authentication and key exchange, a 48-byte pre_master_secret is generated by the client, encrypted under the server's public key, and sent to the server. The server uses its private key to decrypt the pre_master_secret. Both parties then convert the pre_master_secret into the master_secret, as specified above.

RSA digital signatures are performed using PKCS #1 [PKCS1] block type 1. RSA public key encryption is performed using PKCS #1 block type 2.

8.1.2. Diffie-Hellman

A conventional Diffie-Hellman computation is performed. The negotiated key (Z) is used as the pre_master_secret, and is converted into the master_secret, as specified above. Leading bytes of Z that contain all zero bits are stripped before it is used as the pre_master_secret.

Note: Diffie-Hellman parameters are specified by the server, and may be either ephemeral or contained within the server's certificate.

9. Mandatory Cipher Suites

In the absence of an application profile standard specifying otherwise, a TLS compliant application MUST implement the cipher suite TLS_RSA_WITH_3DES_EDE_CBC_SHA.

10. Application data protocol

Application data messages are carried by the Record Layer and are fragmented, compressed and encrypted based on the current connection state. The messages are treated as transparent data to the record layer.

11. IANA Considerations

This document describes a number of new registries to be created by IANA. We recommend that they be placed as individual registries items under a common TLS category.

Section 7.4.5 describes a TLS HashType Registry to be maintained by the IANA, as defining a number of such code point identifiers. HashType identifiers with values in the range 0-63 (decimal) inclusive are assigned via RFC 2434 Standards Action. Values from the range 64-223 (decimal) inclusive are assigned via [RFC 2434] Specification Required. Identifier values from 224-255 (decimal)

inclusive are reserved for RFC 2434 Private Use. The registry will be initially populated with the values in this document, Section 7.4.5.

Section 7.4.5 describes a TLS ClientCertificateType Registry to be maintained by the IANA, as defining a number of such code point identifiers. ClientCertificateType identifiers with values in the range 0-63 (decimal) inclusive are assigned via RFC 2434 Standards Action. Values from the range 64-223 (decimal) inclusive are assigned via [RFC 2434] Specification Required. Identifier values from 224-255 (decimal) inclusive are reserved for RFC 2434 Private Use. The registry will be initially populated with the values in this document, Section 7.4.5.

Section A.5 describes a TLS Cipher Suite Registry to be maintained by the IANA, as well as defining a number of such cipher suite identifiers. Cipher suite values with the first byte in the range 0-191 (decimal) inclusive are assigned via RFC 2434 Standards Action. Values with the first byte in the range 192-254 (decimal) are assigned via RFC 2434 Specification Required. Values with the first byte 255 (decimal) are reserved for RFC 2434 Private Use. The registry will be initially populated with the values from Section A.5 of this document, [TLSAES], and Section 3 of [TLSKRB].

Section 6 requires that all ContentType values be defined by RFC 2434 Standards Action. IANA SHOULD create a TLS ContentType registry, initially populated with values from Section 6.2.1 of this document. Future values MUST be allocated via Standards Action as described in [RFC 2434].

Section 7.2.2 requires that all Alert values be defined by RFC 2434 Standards Action. IANA SHOULD create a TLS Alert registry, initially populated with values from Section 7.2 of this document and Section 4 of [TLSEXT]. Future values MUST be allocated via Standards Action as described in [RFC 2434].

Section 7.4 requires that all HandshakeType values be defined by RFC 2434 Standards Action. IANA SHOULD create a TLS HandshakeType registry, initially populated with values from Section 7.4 of this document and Section 2.4 of [TLSEXT]. Future values MUST be allocated via Standards Action as described in [RFC2434].

11.1 Extensions

Sections XXX and XXX describes a registry of ExtensionType values to be maintained by the IANA. ExtensionType values are to be assigned via IETF Consensus as defined in RFC 2434 [IANA]. The initial registry corresponds to the definition of "ExtensionType" in Section

2.3.

The MIME type "application/pkix-pkipath" has been registered by the IANA with the following template:

```
   To: ietf-types@iana.org Subject: Registration of MIME media type
   application/pkix-pkipath

   MIME media type name: application
   MIME subtype name: pkix-pkipath

   Optional parameters: version (default value is "1")

   Encoding considerations:
      This MIME type is a DER encoding of the ASN.1 type PkiPath,
      defined as follows:
        PkiPath ::= SEQUENCE OF Certificate
        PkiPath is used to represent a certification path.  Within the
        sequence, the order of certificates is such that the subject of
        the first certificate is the issuer of the second certificate,
        etc.

      This is identical to the definition published in [X509-4th-TC1];
      note that it is different from that in [X509-4th].

      All Certificates MUST conform to [PKIX].  (This should be
      interpreted as a requirement to encode only PKIX-conformant
      certificates using this type.  It does not necessarily require
      that all certificates that are not strictly PKIX-conformant must
      be rejected by relying parties, although the security consequences
      of accepting any such certificates should be considered
      carefully.)

      DER (as opposed to BER) encoding MUST be used.  If this type is
      sent over a 7-bit transport, base64 encoding SHOULD be used.

   Security considerations:
      The security considerations of [X509-4th] and [PKIX] (or any
      updates to them) apply, as well as those of any protocol that uses
      this type (e.g., TLS).

      Note that this type only specifies a certificate chain that can be
      assessed for validity according to the relying party's existing
      configuration of trusted CAs; it is not intended to be used to
      specify any change to that configuration.

   Interoperability considerations:
      No specific interoperability problems are known with this type,
      but for recommendations relating to X.509 certificates in general,
      see [PKIX].

   Published specification: this memo, and [PKIX].

   Applications which use this media type: TLS.  It may also be used by
      other protocols, or for general interchange of PKIX certificate

   Additional information:
      Magic number(s): DER-encoded ASN.1 can be easily recognized.
        Further parsing is required to distinguish from other ASN.1
        types.
      File extension(s): .pkipath
      Macintosh File Type Code(s): not specified

   Person & email address to contact for further information:
      Magnus Nystrom <magnus@rsasecurity.com>

   Intended usage: COMMON

   Change controller:
      IESG <iesg@ietf.org>
```

A. Protocol constant values

This section describes protocol types and constants.

A.1. Record layer

```
    struct {
        uint8 major, minor;
    } ProtocolVersion;

    ProtocolVersion version = { 3, 2 };     /* TLS v1.1 */

    enum {
        change_cipher_spec(20), alert(21), handshake(22),
        application_data(23), (255)
    } ContentType;

    struct {
        ContentType type;
        ProtocolVersion version;
        uint16 length;
        opaque fragment[TLSPlaintext.length];
    } TLSPlaintext;

    struct {
        ContentType type;
        ProtocolVersion version;
        uint16 length;
        opaque fragment[TLSCompressed.length];
    } TLSCompressed;

    struct {
        ContentType type;
        ProtocolVersion version;
        uint16 length;
        select (CipherSpec.cipher_type) {
            case stream: GenericStreamCipher;
            case block:  GenericBlockCipher;
        } fragment;
    } TLSCiphertext;

    stream-ciphered struct {
        opaque content[TLSCompressed.length];
        opaque MAC[CipherSpec.hash_size];
    } GenericStreamCipher;

    block-ciphered struct {
        opaque IV[CipherSpec.block_length];
```

```
        opaque content[TLSCompressed.length];
        opaque MAC[CipherSpec.hash_size];
        uint8 padding[GenericBlockCipher.padding_length];
        uint8 padding_length;
    } GenericBlockCipher;
```

A.2. Change cipher specs message

```
    struct {
        enum { change_cipher_spec(1), (255) } type;
    } ChangeCipherSpec;
```

A.3. Alert messages

```
    enum { warning(1), fatal(2), (255) } AlertLevel;

        enum {
            close_notify(0),
            unexpected_message(10),
            bad_record_mac(20),
            decryption_failed(21),
            record_overflow(22),
            decompression_failure(30),
            handshake_failure(40),
            no_certificate_RESERVED (41),
            bad_certificate(42),
            unsupported_certificate(43),
            certificate_revoked(44),
            certificate_expired(45),
            certificate_unknown(46),
            illegal_parameter(47),
            unknown_ca(48),
            access_denied(49),
            decode_error(50),
            decrypt_error(51),
            export_restriction_RESERVED(60),
            protocol_version(70),
            insufficient_security(71),
            internal_error(80),
            user_canceled(90),
            no_renegotiation(100),
            unsupported_extension(110),           /* new */
            certificate_unobtainable(111),        /* new */
            unrecognized_name(112),               /* new */
            bad_certificate_status_response(113), /* new */
            bad_certificate_hash_value(114),      /* new */
            (255)
        } AlertDescription;
```

```
    struct {
        AlertLevel level;
        AlertDescription description;
    } Alert;
```

A.4. Handshake protocol

```
    enum {
        hello_request(0), client_hello(1), server_hello(2),
        certificate(11), server_key_exchange (12),
        certificate_request(13), server_hello_done(14),
        certificate_verify(15), client_key_exchange(16),
        finished(20), certificate_url(21), certificate_status(22),
     (255)
    } HandshakeType;

    struct {
        HandshakeType msg_type;
        uint24 length;
        select (HandshakeType) {
            case hello_request:       HelloRequest;
            case client_hello:        ClientHello;
            case server_hello:        ServerHello;
            case certificate:         Certificate;
            case server_key_exchange: ServerKeyExchange;
            case certificate_request: CertificateRequest;
            case server_hello_done:   ServerHelloDone;
            case certificate_verify:  CertificateVerify;
            case client_key_exchange: ClientKeyExchange;
            case finished:            Finished;
            case certificate_url:     CertificateURL;
            case certificate_status:  CertificateStatus;
        } body;
    } Handshake;
```

A.4.1. Hello messages

```
    struct { } HelloRequest;

    struct {
        uint32 gmt_unix_time;
        opaque random_bytes[28];
    } Random;

    opaque SessionID<0..32>;

    uint8 CipherSuite[2];

    enum { null(0), (255) } CompressionMethod;

    struct {
        ProtocolVersion client_version;
        Random random;
```

```
        SessionID session_id;
        CipherSuite cipher_suites<2..2^16-1>;
        CompressionMethod compression_methods<1..2^8-1>;
        Extension client_hello_extension_list<0..2^16-1>;
    } ClientHello;

    struct {
        ProtocolVersion client_version;
        Random random;
        SessionID session_id;
        CipherSuite cipher_suites<2..2^16-1>;
        CompressionMethod compression_methods<1..2^8-1>;
        Extension client_hello_extension_list<0..2^16-1>;
    } ExtendedClientHello;

    struct {
        ProtocolVersion server_version;
        Random random;
        SessionID session_id;
        CipherSuite cipher_suite;
        CompressionMethod compression_method;
    } ServerHello;

    struct {
        ProtocolVersion server_version;
        Random random;
        SessionID session_id;
        CipherSuite cipher_suite;
        CompressionMethod compression_method;
     Extension server_hello_extension_list<0..2^16-1>;
    } ExtendedServerHello;
```

```
     struct {
         ExtensionType extension_type;
         opaque extension_data<0..2^16-1>;
     } Extension;

     enum {
         server_name(0), max_fragment_length(1),
         client_certificate_url(2), trusted_ca_keys(3),
         truncated_hmac(4), status_request(5),
         cert_hash_types(6), (65535)
     } ExtensionType;

     struct {
         NameType name_type;
         select (name_type) {
             case host_name: HostName;
```

```
         } name;
     } ServerName;

     enum {
         host_name(0), (255)
     } NameType;

     opaque HostName<1..2^16-1>;

     struct {
         ServerName server_name_list<1..2^16-1>
     } ServerNameList;

     enum{
         2^9(1), 2^10(2), 2^11(3), 2^12(4), (255)
     } MaxFragmentLength;

     struct {
         TrustedAuthority trusted_authorities_list<0..2^16-1>;
     } TrustedAuthorities;

     struct {
         IdentifierType identifier_type;
         select (identifier_type) {
             case pre_agreed: struct {};
             case key_sha1_hash: SHA1Hash;
             case x509_name: DistinguishedName;
             case cert_sha1_hash: SHA1Hash;
         } identifier;
```

```
    } TrustedAuthority;

    enum {
        pre_agreed(0), key_sha1_hash(1), x509_name(2),
        cert_sha1_hash(3), (255)
    } IdentifierType;

    struct {
        CertificateStatusType status_type;
        select (status_type) {
            case ocsp: OCSPStatusRequest;
        } request;
    } CertificateStatusRequest;

    enum { ocsp(1), (255) } CertificateStatusType;

    struct {
        ResponderID responder_id_list<0..2^16-1>;
        Extensions  request_extensions;
```

```
    } OCSPStatusRequest;

     opaque ResponderID<1..2^16-1>;
```

A.4.2. Server authentication and key exchange messages

```
    opaque ASN.1Cert<2^24-1>;

    struct {
        ASN.1Cert certificate_list<0..2^24-1>;
    } Certificate;

    struct {
        CertificateStatusType status_type;
        select (status_type) {
            case ocsp: OCSPResponse;
        } response;
    } CertificateStatus;

    opaque OCSPResponse<1..2^24-1>;

    enum { rsa, diffie_hellman } KeyExchangeAlgorithm;

    struct {
        opaque rsa_modulus<1..2^16-1>;
        opaque rsa_exponent<1..2^16-1>;
    } ServerRSAParams;
    struct {
        opaque dh_p<1..2^16-1>;
        opaque dh_g<1..2^16-1>;
        opaque dh_Ys<1..2^16-1>;
    } ServerDHParams;

    struct {
        select (KeyExchangeAlgorithm) {
            case diffie_hellman:
                ServerDHParams params;
                Signature signed_params;
            case rsa:
                ServerRSAParams params;
                Signature signed_params;
        };
    } ServerKeyExchange;

    enum { anonymous, rsa, dsa } SignatureAlgorithm;

    struct {
        select (KeyExchangeAlgorithm) {
```

```
            case diffie_hellman:
                ServerDHParams params;
            case rsa:
                ServerRSAParams params;
        };
    } ServerParams;

    struct {
        select (SignatureAlgorithm) {
            case anonymous: struct { };
            case rsa:
                digitally-signed struct {
                    opaque hash[Hash.length];
                };
            case dsa:
                digitally-signed struct {
                    opaque sha_hash[20];
                };
            };
        };
    } Signature;

    enum {
        rsa_sign(1), dss_sign(2), rsa_fixed_dh(3), dss_fixed_dh(4),
     rsa_ephemeral_dh_RESERVED(5), dss_ephemeral_dh_RESERVED(6),
     fortezza_dms_RESERVED(20),
     (255)
    } ClientCertificateType;

    opaque DistinguishedName<1..2^16-1>;

    struct {
        ClientCertificateType certificate_types<1..2^8-1>;
        DistinguishedName certificate_authorities<0..2^16-1>;
    } CertificateRequest;

    struct { } ServerHelloDone;
```

A.4.3. Client authentication and key exchange messages

```
       struct {
           select (KeyExchangeAlgorithm) {
               case rsa: EncryptedPreMasterSecret;
               case diffie_hellman: ClientDiffieHellmanPublic;
           } exchange_keys;
       } ClientKeyExchange;

       struct {
```

```
           ProtocolVersion client_version;
           opaque random[46];
       } PreMasterSecret;

       struct {
           public-key-encrypted PreMasterSecret pre_master_secret;
       } EncryptedPreMasterSecret;

       enum { implicit, explicit } PublicValueEncoding;

       struct {
           select (PublicValueEncoding) {
               case implicit: struct {};
               case explicit: opaque DH_Yc<1..2^16-1>;
           } dh_public;
       } ClientDiffieHellmanPublic;

       enum {
           individual_certs(0), pkipath(1), (255)
       } CertChainType;
```

```
    enum {
        false(0), true(1)
    } Boolean;

    struct {
        CertChainType type;
        URLAndOptionalHash url_and_hash_list<1..2^16-1>;
    } CertificateURL;

    struct {
        opaque url<1..2^16-1>;
        Boolean hash_present;
        select (hash_present) {
            case false: struct {};
            case true: SHA1Hash;
        } hash;
    } URLAndOptionalHash;

    opaque SHA1Hash[20];

    struct {
        Signature signature;
    } CertificateVerify;
```

A.4.4. Handshake finalization message

```
    struct {
```

```
        opaque verify_data[12];
    } Finished;
```

A.5. The CipherSuite

The following values define the CipherSuite codes used in the client hello and server hello messages.

A CipherSuite defines a cipher specification supported in TLS Version 1.1.

TLS_NULL_WITH_NULL_NULL is specified and is the initial state of a TLS connection during the first handshake on that channel, but must not be negotiated, as it provides no more protection than an unsecured connection.

```
    CipherSuite TLS_NULL_WITH_NULL_NULL                = { 0x00,0x00 };
```

The following CipherSuite definitions require that the server provide an RSA certificate that can be used for key exchange. The server may request either an RSA or a DSS signature-capable certificate in the certificate request message.

```
 CipherSuite TLS_RSA_WITH_NULL_MD5                    = { 0x00,0x01 };
 CipherSuite TLS_RSA_WITH_NULL_SHA                    = { 0x00,0x02 };
 CipherSuite TLS_RSA_WITH_RC4_128_MD5                 = { 0x00,0x04 };
 CipherSuite TLS_RSA_WITH_RC4_128_SHA                 = { 0x00,0x05 };
 CipherSuite TLS_RSA_WITH_IDEA_CBC_SHA                = { 0x00,0x07 };
 CipherSuite TLS_RSA_WITH_DES_CBC_SHA                 = { 0x00,0x09 };
 CipherSuite TLS_RSA_WITH_3DES_EDE_CBC_SHA            = { 0x00,0x0A };
 CipherSuite TLS_RSA_WITH_AES_128_CBC_SHA             = { 0x00, 0x2F };
 CipherSuite TLS_RSA_WITH_AES_256_CBC_SHA             = { 0x00, 0x35 };
```

The following CipherSuite definitions are used for server-authenticated (and optionally client-authenticated) Diffie-Hellman. DH denotes cipher suites in which the server's certificate contains the Diffie-Hellman parameters signed by the certificate authority (CA). DHE denotes ephemeral Diffie-Hellman, where the Diffie-Hellman parameters are signed by a DSS or RSA certificate, which has been signed by the CA. The signing algorithm used is specified after the DH or DHE parameter. The server can request an RSA or DSS signature-capable certificate from the client for client authentication or it may request a Diffie-Hellman certificate. Any Diffie-Hellman certificate provided by the client must use the parameters (group and generator) described by the server.

```
 CipherSuite TLS_DH_DSS_WITH_DES_CBC_SHA              = { 0x00,0x0C };
 CipherSuite TLS_DH_DSS_WITH_3DES_EDE_CBC_SHA         = { 0x00,0x0D };
 CipherSuite TLS_DH_RSA_WITH_DES_CBC_SHA              = { 0x00,0x0F };
```

```
 CipherSuite TLS_DH_RSA_WITH_3DES_EDE_CBC_SHA         = { 0x00,0x10 };
 CipherSuite TLS_DHE_DSS_WITH_DES_CBC_SHA             = { 0x00,0x12 };
 CipherSuite TLS_DHE_DSS_WITH_3DES_EDE_CBC_SHA        = { 0x00,0x13 };
 CipherSuite TLS_DHE_RSA_WITH_DES_CBC_SHA             = { 0x00,0x15 };
 CipherSuite TLS_DHE_RSA_WITH_3DES_EDE_CBC_SHA        = { 0x00,0x16 };
 CipherSuite TLS_DH_DSS_WITH_AES_128_CBC_SHA          = { 0x00, 0x30 };
 CipherSuite TLS_DH_RSA_WITH_AES_128_CBC_SHA          = { 0x00, 0x31 };
 CipherSuite TLS_DHE_DSS_WITH_AES_128_CBC_SHA         = { 0x00, 0x32 };
 CipherSuite TLS_DHE_RSA_WITH_AES_128_CBC_SHA         = { 0x00, 0x33 };
 CipherSuite TLS_DH_anon_WITH_AES_128_CBC_SHA         = { 0x00, 0x34 };
 CipherSuite TLS_DH_DSS_WITH_AES_256_CBC_SHA          = { 0x00, 0x36 };
 CipherSuite TLS_DH_RSA_WITH_AES_256_CBC_SHA          = { 0x00, 0x37 };
 CipherSuite TLS_DHE_DSS_WITH_AES_256_CBC_SHA         = { 0x00, 0x38 };
 CipherSuite TLS_DHE_RSA_WITH_AES_256_CBC_SHA         = { 0x00, 0x39 };
 CipherSuite TLS_DH_anon_WITH_AES_256_CBC_SHA       = { 0x00, 0x3A };
```

The following cipher suites are used for completely anonymous Diffie-Hellman communications in which neither party is authenticated. Note that this mode is vulnerable to man-in-the-middle attacks and is therefore deprecated.

```
 CipherSuite TLS_DH_anon_WITH_RC4_128_MD5           = { 0x00,0x18 };
 CipherSuite TLS_DH_anon_WITH_DES_CBC_SHA           = { 0x00,0x1A };
 CipherSuite TLS_DH_anon_WITH_3DES_EDE_CBC_SHA      = { 0x00,0x1B };
```

When SSLv3 and TLS 1.0 were designed, the United States restricted the export of cryptographic software containing certain strong encryption algorithms. A series of cipher suites were designed to operate at reduced key lengths in order to comply with those regulations. Due to advances in computer performance, these algorithms are now unacceptably weak and export restrictions have since been loosened. TLS 1.1 implementations MUST NOT negotiate these cipher suites in TLS 1.1 mode. However, for backward compatibility they may be offered in the ClientHello for use with TLS 1.0 or SSLv3 only servers. TLS 1.1 clients MUST check that the server did not choose one of these cipher suites during the handshake. These ciphersuites are listed below for informational purposes and to reserve the numbers.

```
 CipherSuite TLS_RSA_EXPORT_WITH_RC4_40_MD5         = { 0x00,0x03 };
 CipherSuite TLS_RSA_EXPORT_WITH_RC2_CBC_40_MD5     = { 0x00,0x06 };
 CipherSuite TLS_RSA_EXPORT_WITH_DES40_CBC_SHA      = { 0x00,0x08 };
 CipherSuite TLS_DH_DSS_EXPORT_WITH_DES40_CBC_SHA   = { 0x00,0x0B };
 CipherSuite TLS_DH_RSA_EXPORT_WITH_DES40_CBC_SHA   = { 0x00,0x0E };
 CipherSuite TLS_DHE_DSS_EXPORT_WITH_DES40_CBC_SHA  = { 0x00,0x11 };
 CipherSuite TLS_DHE_RSA_EXPORT_WITH_DES40_CBC_SHA  = { 0x00,0x14 };
 CipherSuite TLS_DH_anon_EXPORT_WITH_RC4_40_MD5     = { 0x00,0x17 };
 CipherSuite TLS_DH_anon_EXPORT_WITH_DES40_CBC_SHA  = { 0x00,0x19 };
```

The following cipher suites were defined in [TLSKRB] and are included here for completeness. See [TLSKRB] for details:

```
 CipherSuite      TLS_KRB5_WITH_DES_CBC_SHA             = { 0x00,0x1E };
 CipherSuite      TLS_KRB5_WITH_3DES_EDE_CBC_SHA        = { 0x00,0x1F };
 CipherSuite      TLS_KRB5_WITH_RC4_128_SHA             = { 0x00,0x20 };
 CipherSuite      TLS_KRB5_WITH_IDEA_CBC_SHA            = { 0x00,0x21 };
 CipherSuite      TLS_KRB5_WITH_DES_CBC_MD5             = { 0x00,0x22 };
 CipherSuite      TLS_KRB5_WITH_3DES_EDE_CBC_MD5        = { 0x00,0x23 };
 CipherSuite      TLS_KRB5_WITH_RC4_128_MD5             = { 0x00,0x24 };
 CipherSuite      TLS_KRB5_WITH_IDEA_CBC_MD5            = { 0x00,0x25 };
```

The following exportable cipher suites were defined in [TLSKRB] and are included here for completeness. TLS 1.1 implementations MUST NOT negotiate these cipher suites.

```
 CipherSuite      TLS_KRB5_EXPORT_WITH_DES_CBC_40_SHA  = { 0x00,0x26
};
 CipherSuite      TLS_KRB5_EXPORT_WITH_RC2_CBC_40_SHA  = { 0x00,0x27
};
 CipherSuite      TLS_KRB5_EXPORT_WITH_RC4_40_SHA      = { 0x00,0x28
};
 CipherSuite      TLS_KRB5_EXPORT_WITH_DES_CBC_40_MD5  = { 0x00,0x29
};
 CipherSuite      TLS_KRB5_EXPORT_WITH_RC2_CBC_40_MD5  = { 0x00,0x2A
};
 CipherSuite      TLS_KRB5_EXPORT_WITH_RC4_40_MD5      = { 0x00,0x2B
};
```

The cipher suite space is divided into three regions:

1. Cipher suite values with first byte 0x00 (zero) through decimal 191 (0xBF) inclusive are reserved for the IETF Standards Track protocols.

2. Cipher suite values with first byte decimal 192 (0xC0) through decimal 254 (0xFE) inclusive are reserved for assignment for non-Standards Track methods.

3. Cipher suite values with first byte 0xFF are reserved for private use.

Additional information describing the role of IANA in the allocation of cipher suite code points is described in Section 11.

Note: The cipher suite values { 0x00, 0x1C } and { 0x00, 0x1D } are reserved to avoid collision with Fortezza-based cipher suites in SSL 3.

A.6. The Security Parameters

These security parameters are determined by the TLS Handshake Protocol and provided as parameters to the TLS Record Layer in order to initialize a connection state. SecurityParameters includes:

```
    enum { null(0), (255) } CompressionMethod;
```

```
       enum { server, client } ConnectionEnd;

       enum { null, rc4, rc2, des, 3des, des40, aes, idea }
       BulkCipherAlgorithm;

       enum { stream, block } CipherType;

       enum { null, md5, sha } MACAlgorithm;

   /* The algorithms specified in CompressionMethod,
   BulkCipherAlgorithm, and MACAlgorithm may be added to. */

       struct {
           ConnectionEnd entity;
           BulkCipherAlgorithm bulk_cipher_algorithm;
           CipherType cipher_type;
           uint8 key_size;
           uint8 key_material_length;
           MACAlgorithm mac_algorithm;
           uint8 hash_size;
           CompressionMethod compression_algorithm;
           opaque master_secret[48];
           opaque client_random[32];
           opaque server_random[32];
       } SecurityParameters;
```

B. Glossary

   Advanced Encryption Standard (AES)
       AES is a widely used symmetric encryption algorithm.
       AES is

a block cipher with a 128, 192, or 256 bit keys and a 16 byte block size. [AES] TLS currently only supports the 128 and 256 bit key sizes.

application protocol
An application protocol is a protocol that normally layers directly on top of the transport layer (e.g., TCP/IP). Examples include HTTP, TELNET, FTP, and SMTP.

asymmetric cipher
See public key cryptography.

authentication
Authentication is the ability of one entity to determine the identity of another entity.

block cipher
A block cipher is an algorithm that operates on plaintext in groups of bits, called blocks. 64 bits is a common block size.

bulk cipher
A symmetric encryption algorithm used to encrypt large quantities of data.

cipher block chaining (CBC)
CBC is a mode in which every plaintext block encrypted with a block cipher is first exclusive-ORed with the previous ciphertext block (or, in the case of the first block, with the initialization vector). For decryption, every block is first decrypted, then exclusive-ORed with the previous ciphertext block (or IV).

certificate
As part of the X.509 protocol (a.k.a. ISO Authentication framework), certificates are assigned by a trusted Certificate Authority and provide a strong binding between a party's identity or some other attributes and its public key.

client
The application entity that initiates a TLS connection to a server. This may or may not imply that the client initiated the underlying transport connection. The primary operational difference between the server and client is that the server is

generally authenticated, while the client is only optionally authenticated.

client write key
    The key used to encrypt data written by the client.

client write MAC secret
    The secret data used to authenticate data written by the client.

connection
    A connection is a transport (in the OSI layering model definition) that provides a suitable type of service. For TLS, such connections are peer to peer relationships. The connections are transient. Every connection is associated with one session.

Data Encryption Standard
    DES is a very widely used symmetric encryption algorithm. DES is a block cipher with a 56 bit key and an 8 byte block size. Note that in TLS, for key generation purposes, DES is treated as having an 8 byte key length (64 bits), but it still only provides 56 bits of protection. (The low bit of each key byte is presumed to be set to produce odd parity in that key byte.) DES can also be operated in a mode where three independent keys and three encryptions are used for each block of data; this uses 168 bits of key (24 bytes in the TLS key generation method) and provides the equivalent of 112 bits of security. [DES], [3DES]

Digital Signature Standard (DSS)
    A standard for digital signing, including the Digital Signing Algorithm, approved by the National Institute of Standards and Technology, defined in NIST FIPS PUB 186, "Digital Signature Standard," published May, 1994 by the U.S. Dept. of Commerce. [DSS]

digital signatures
    Digital signatures utilize public key cryptography and one-way hash functions to produce a signature of the data that can be authenticated, and is difficult to forge or repudiate.

handshake
    An initial negotiation between client and server that establishes the parameters of their transactions.

Initialization Vector (IV)
    When a block cipher is used in CBC mode, the initialization vector is exclusive-ORed with the first plaintext block prior to encryption.

IDEA
    A 64-bit block cipher designed by Xuejia Lai and James Massey.
    [IDEA]

Message Authentication Code (MAC)
    A Message Authentication Code is a one-way hash computed from a
    message and some secret data. It is difficult to forge without
    knowing the secret data. Its purpose is to detect if the message
    has been altered.

master secret
    Secure secret data used for generating encryption keys, MAC
    secrets, and IVs.

MD5
    MD5 is a secure hashing function that converts an arbitrarily
    long data stream into a digest of fixed size (16 bytes). [MD5]

public key cryptography
    A class of cryptographic techniques employing two-key ciphers.
    Messages encrypted with the public key can only be decrypted with
    the associated private key. Conversely, messages signed with the
    private key can be verified with the public key.

one-way hash function
    A one-way transformation that converts an arbitrary amount of
    data into a fixed-length hash. It is computationally hard to
    reverse the transformation or to find collisions. MD5 and SHA are
    examples of one-way hash functions.

RC2
    A block cipher developed by Ron Rivest at RSA Data Security, Inc.
    [RSADSI] described in [RC2].

RC4
    A stream cipher invented by Ron Rivest. A compatible cipher is
    described in [SCH].

RSA
    A very widely used public-key algorithm that can be used for
    either encryption or digital signing. [RSA]

server
    The server is the application entity that responds to requests
    for connections from clients. See also under client.

session
    A TLS session is an association between a client and a server.
    Sessions are created by the handshake protocol. Sessions define a
    set of cryptographic security parameters, which can be shared
    among multiple connections. Sessions are used to avoid the
    expensive negotiation of new security parameters for each
    connection.

session identifier
    A session identifier is a value generated by a server that
    identifies a particular session.

server write key
    The key used to encrypt data written by the server.

server write MAC secret
    The secret data used to authenticate data written by the server.

SHA
    The Secure Hash Algorithm is defined in FIPS PUB 180-2. It
    produces a 20-byte output. Note that all references to SHA
    actually use the modified SHA-1 algorithm. [SHA]

SSL
    Netscape's Secure Socket Layer protocol [SSL3]. TLS is based on
    SSL Version 3.0

stream cipher
    An encryption algorithm that converts a key into a
    cryptographically-strong keystream, which is then exclusive-ORed
    with the plaintext.

symmetric cipher
    See bulk cipher.

Transport Layer Security (TLS)
    This protocol; also, the Transport Layer Security working group
    of the Internet Engineering Task Force (IETF). See "Comments" at
    the end of this document.

C. CipherSuite definitions

CipherSuite	Key Exchange	Cipher	Hash
TLS_NULL_WITH_NULL_NULL	NULL	NULL	NULL
TLS_RSA_WITH_NULL_MD5	RSA	NULL	MD5
TLS_RSA_WITH_NULL_SHA	RSA	NULL	SHA
TLS_RSA_WITH_RC4_128_MD5	RSA	RC4_128	MD5
TLS_RSA_WITH_RC4_128_SHA	RSA	RC4_128	SHA
TLS_RSA_WITH_IDEA_CBC_SHA	RSA	IDEA_CBC	SHA
TLS_RSA_WITH_DES_CBC_SHA	RSA	DES_CBC	SHA
TLS_RSA_WITH_3DES_EDE_CBC_SHA	RSA	3DES_EDE_CBC	SHA
TLS_RSA_WITH_AES_128_CBC_SHA	RSA	AES_128_CBC	SHA
TLS_RSA_WITH_AES_256_SHA	RSA	AES_256_CBC	SHA
TLS_DH_DSS_WITH_DES_CBC_SHA	DH_DSS	DES_CBC	SHA
TLS_DH_DSS_WITH_3DES_EDE_CBC_SHA	DH_DSS	3DES_EDE_CBC	SHA
TLS_DH_RSA_WITH_DES_CBC_SHA	DH_RSA	DES_CBC	SHA
TLS_DH_RSA_WITH_3DES_EDE_CBC_SHA	DH_RSA	3DES_EDE_CBC	SHA
TLS_DHE_DSS_WITH_DES_CBC_SHA	DHE_DSS	DES_CBC	SHA
TLS_DHE_DSS_WITH_3DES_EDE_CBC_SHA	DHE_DSS	3DES_EDE_CBC	SHA
TLS_DHE_RSA_WITH_DES_CBC_SHA	DHE_RSA	DES_CBC	SHA
TLS_DHE_RSA_WITH_3DES_EDE_CBC_SHA	DHE_RSA	3DES_EDE_CBC	SHA
TLS_DH_anon_WITH_RC4_128_MD5	DH_anon	RC4_128	MD5
TLS_DH_anon_WITH_DES_CBC_SHA	DH_anon	DES_CBC	SHA
TLS_DH_anon_WITH_3DES_EDE_CBC_SHA	DH_anon	3DES_EDE_CBC	SHA
TLS_DH_DSS_WITH_AES_128_CBC_SHA	DH_DSS	AES_128_CBC	SHA
TLS_DH_RSA_WITH_AES_128_CBC_SHA	DH_RSA	AES_128_CBC	SHA
TLS_DHE_DSS_WITH_AES_128_CBC_SHA	DHE_DSS	AES_128_CBC	SHA
TLS_DHE_RSA_WITH_AES_128_CBC_SHA	DHE_RSA	AES_128_CBC	SHA
TLS_DH_anon_WITH_AES_128_CBC_SHA	DH_anon	AES_128_CBC	SHA
TLS_DH_DSS_WITH_AES_256_CBC_SHA	DH_DSS	AES_256_CBC	SHA
TLS_DH_RSA_WITH_AES_256_CBC_SHA	DH_RSA	AES_256_CBC	SHA
TLS_DHE_DSS_WITH_AES_256_CBC_SHA	DHE_DSS	AES_256_CBC	SHA
TLS_DHE_RSA_WITH_AES_256_CBC_SHA	DHE_RSA	AES_256_CBC	SHA
TLS_DH_anon_WITH_AES_256_CBC_SHA	DH_anon	AES_256_CBC	SHA

Key Exchange Algorithm	Description	Key size limit
DHE_DSS	Ephemeral DH with DSS signatures	None
DHE_RSA	Ephemeral DH with RSA signatures	None
DH_anon	Anonymous DH, no signatures	None

DH_DSS	DH with DSS-based certificates	None
DH_RSA	DH with RSA-based certificates	None
		RSA = none
NULL	No key exchange	N/A

RSA	RSA key exchange	None

Cipher	Type	Key Material	Expanded Key Material	IV Size	Block Size
NULL	Stream	0	0	0	N/A
IDEA_CBC	Block	16	16	8	8
RC2_CBC_40	Block	5	16	8	8
RC4_40	Stream	5	16	0	N/A
RC4_128	Stream	16	16	0	N/A
DES40_CBC	Block	5	8	8	8
DES_CBC	Block	8	8	8	8
3DES_EDE_CBC	Block	24	24	8	8

Type
Indicates whether this is a stream cipher or a block cipher running in CBC mode.

Key Material
The number of bytes from the key_block that are used for generating the write keys.

Expanded Key Material
The number of bytes actually fed into the encryption algorithm

IV Size
How much data needs to be generated for the initialization vector. Zero for stream ciphers; equal to the block size for block ciphers.

Block Size
The amount of data a block cipher enciphers in one chunk; a block cipher running in CBC mode can only encrypt an even multiple of its block size.

Hash function	Hash Size	Padding Size
NULL	0	0
MD5	16	48
SHA	20	40

D. Implementation Notes

The TLS protocol cannot prevent many common security mistakes. This section provides several recommendations to assist implementors.

D.1 Random Number Generation and Seeding

TLS requires a cryptographically-secure pseudorandom number generator (PRNG). Care must be taken in designing and seeding PRNGs. PRNGs based on secure hash operations, most notably MD5 and/or SHA, are acceptable, but cannot provide more security than the size of the random number generator state. (For example, MD5-based PRNGs usually provide 128 bits of state.)

To estimate the amount of seed material being produced, add the number of bits of unpredictable information in each seed byte. For example, keystroke timing values taken from a PC compatible's 18.2 Hz timer provide 1 or 2 secure bits each, even though the total size of the counter value is 16 bits or more. To seed a 128-bit PRNG, one would thus require approximately 100 such timer values.

[RANDOM] provides guidance on the generation of random values.

D.2 Certificates and authentication

Implementations are responsible for verifying the integrity of certificates and should generally support certificate revocation messages. Certificates should always be verified to ensure proper signing by a trusted Certificate Authority (CA). The selection and addition of trusted CAs should be done very carefully. Users should be able to view information about the certificate and root CA.

D.3 CipherSuites

TLS supports a range of key sizes and security levels, including some which provide no or minimal security. A proper implementation will probably not support many cipher suites. For example, 40-bit encryption is easily broken, so implementations requiring strong security should not allow 40-bit keys. Similarly, anonymous Diffie-Hellman is strongly discouraged because it cannot prevent man-in-the-middle attacks. Applications should also enforce minimum and maximum key sizes. For example, certificate chains containing 512-bit RSA keys or signatures are not appropriate for high-security applications.

E. Backward Compatibility With SSL

For historical reasons and in order to avoid a profligate consumption of reserved port numbers, application protocols which are secured by TLS 1.1, TLS 1.0, SSL 3.0, and SSL 2.0 all frequently share the same connection port: for example, the https protocol (HTTP secured by SSL or TLS) uses port 443 regardless of which security protocol it is using. Thus, some mechanism must be determined to distinguish and negotiate among the various protocols.

TLS versions 1.1, 1.0, and SSL 3.0 are very similar; thus, supporting both is easy. TLS clients who wish to negotiate with such older servers SHOULD send client hello messages using the SSL 3.0 record format and client hello structure, sending {3, 2} for the version field to note that they support TLS 1.1. If the server supports only TLS 1.0 or SSL 3.0, it will respond with a downrev 3.0 server hello; if it supports TLS 1.1 it will respond with a TLS 1.1 server hello. The negotiation then proceeds as appropriate for the negotiated protocol.

Similarly, a TLS 1.1 server which wishes to interoperate with TLS 1.0 or SSL 3.0 clients SHOULD accept SSL 3.0 client hello messages and respond with a SSL 3.0 server hello if an SSL 3.0 client hello with a version field of {3, 0} is received, denoting that this client does not support TLS. Similarly, if a SSL 3.0 or TLS 1.0 hello with a version field of {3, 1} is received, the server SHOULD respond with a TLS 1.0 hello with a version field of {3, 1}.

Whenever a client already knows the highest protocol known to a server (for example, when resuming a session), it SHOULD initiate the connection in that native protocol.

TLS 1.1 clients that support SSL Version 2.0 servers MUST send SSL Version 2.0 client hello messages [SSL2]. TLS servers SHOULD accept either client hello format if they wish to support SSL 2.0 clients on the same connection port. The only deviations from the Version 2.0 specification are the ability to specify a version with a value of three and the support for more ciphering types in the CipherSpec.

Warning: The ability to send Version 2.0 client hello messages will be phased out with all due haste. Implementors SHOULD make every effort to move forward as quickly as possible. Version 3.0 provides better mechanisms for moving to newer versions.

The following cipher specifications are carryovers from SSL Version 2.0. These are assumed to use RSA for key exchange and authentication.

```
    V2CipherSpec TLS_RC4_128_WITH_MD5          = { 0x01,0x00,0x80 };
    V2CipherSpec TLS_RC4_128_EXPORT40_WITH_MD5 = { 0x02,0x00,0x80 };
    V2CipherSpec TLS_RC2_CBC_128_CBC_WITH_MD5  = { 0x03,0x00,0x80 };
    V2CipherSpec TLS_RC2_CBC_128_CBC_EXPORT40_WITH_MD5
                                               = { 0x04,0x00,0x80 };
    V2CipherSpec TLS_IDEA_128_CBC_WITH_MD5     = { 0x05,0x00,0x80 };
    V2CipherSpec TLS_DES_64_CBC_WITH_MD5       = { 0x06,0x00,0x40 };
    V2CipherSpec TLS_DES_192_EDE3_CBC_WITH_MD5 = { 0x07,0x00,0xC0 };
```

Cipher specifications native to TLS can be included in Version 2.0 client hello messages using the syntax below. Any V2CipherSpec element with its first byte equal to zero will be ignored by Version 2.0 servers. Clients sending any of the above V2CipherSpecs SHOULD also include the TLS equivalent (see Appendix A.5):

```
    V2CipherSpec (see TLS name) = { 0x00, CipherSuite };
```

Note: TLS 1.2 clients may generate the SSLv2 EXPORT cipher suites in handshakes for backward compatibility but MUST NOT negotiate them in TLS 1.2 mode.

E.1. Version 2 client hello

The Version 2.0 client hello message is presented below using this document's presentation model. The true definition is still assumed to be the SSL Version 2.0 specification. Note that this message MUST be sent directly on the wire, not wrapped as an SSLv3 record

```
    uint8 V2CipherSpec[3];

    struct {
        uint16 msg_length;
```

```
          uint8 msg_type;
          Version version;
          uint16 cipher_spec_length;
          uint16 session_id_length;
          uint16 challenge_length;
          V2CipherSpec cipher_specs[V2ClientHello.cipher_spec_length];
          opaque session_id[V2ClientHello.session_id_length];
          opaque challenge[V2ClientHello.challenge_length;
      } V2ClientHello;
```

msg_length
    This field is the length of the following data in bytes. The high bit MUST be 1 and is not part of the length.

msg_type
    This field, in conjunction with the version field, identifies a

version 2 client hello message. The value SHOULD be one (1).

version
    The highest version of the protocol supported by the client (equals ProtocolVersion.version, see Appendix A.1).

cipher_spec_length
    This field is the total length of the field cipher_specs. It cannot be zero and MUST be a multiple of the V2CipherSpec length (3).

session_id_length
    This field MUST have a value of zero.

challenge_length
    The length in bytes of the client's challenge to the server to authenticate itself. When using the SSLv2 backward compatible handshake the client MUST use a 32-byte challenge.

cipher_specs
    This is a list of all CipherSpecs the client is willing and able to use. There MUST be at least one CipherSpec acceptable to the server.

session_id
    This field MUST be empty.

challenge
    The client challenge to the server for the server to identify

itself is a (nearly) arbitrary length random. The TLS server will right justify the challenge data to become the ClientHello.random data (padded with leading zeroes, if necessary), as specified in this protocol specification. If the length of the challenge is greater than 32 bytes, only the last 32 bytes are used. It is legitimate (but not necessary) for a V3 server to reject a V2 ClientHello that has fewer than 16 bytes of challenge data.

Note: Requests to resume a TLS session MUST use a TLS client hello.

E.2. Avoiding man-in-the-middle version rollback

When TLS clients fall back to Version 2.0 compatibility mode, they SHOULD use special PKCS #1 block formatting. This is done so that TLS servers will reject Version 2.0 sessions with TLS-capable clients.

When TLS clients are in Version 2.0 compatibility mode, they set the right-hand (least-significant) 8 random bytes of the PKCS padding (not including the terminal null of the padding) for the RSA

encryption of the ENCRYPTED-KEY-DATA field of the CLIENT-MASTER-KEY to 0x03 (the other padding bytes are random). After decrypting the ENCRYPTED-KEY-DATA field, servers that support TLS SHOULD issue an error if these eight padding bytes are 0x03. Version 2.0 servers receiving blocks padded in this manner will proceed normally.

F. Security analysis

The TLS protocol is designed to establish a secure connection between a client and a server communicating over an insecure channel. This document makes several traditional assumptions, including that attackers have substantial computational resources and cannot obtain secret information from sources outside the protocol. Attackers are assumed to have the ability to capture, modify, delete, replay, and otherwise tamper with messages sent over the communication channel. This appendix outlines how TLS has been designed to resist a variety of attacks.

F.1. Handshake protocol

The handshake protocol is responsible for selecting a CipherSpec and generating a Master Secret, which together comprise the primary cryptographic parameters associated with a secure session. The handshake protocol can also optionally authenticate parties who have certificates signed by a trusted certificate authority.

F.1.1. Authentication and key exchange

TLS supports three authentication modes: authentication of both parties, server authentication with an unauthenticated client, and total anonymity. Whenever the server is authenticated, the channel is secure against man-in-the-middle attacks, but completely anonymous sessions are inherently vulnerable to such attacks. Anonymous servers cannot authenticate clients. If the server is authenticated, its certificate message must provide a valid certificate chain leading to an acceptable certificate authority. Similarly, authenticated clients must supply an acceptable certificate to the server. Each party is responsible for verifying that the other's certificate is valid and has not expired or been revoked.

The general goal of the key exchange process is to create a pre_master_secret known to the communicating parties and not to attackers. The pre_master_secret will be used to generate the master_secret (see Section 8.1). The master_secret is required to generate the finished messages, encryption keys, and MAC secrets (see Sections 7.4.10, 7.4.11 and 6.3). By sending a correct finished message, parties thus prove that they know the correct pre_master_secret.

F.1.1.1. Anonymous key exchange

Completely anonymous sessions can be established using RSA or Diffie-Hellman for key exchange. With anonymous RSA, the client encrypts a pre_master_secret with the server's uncertified public key extracted

from the server key exchange message. The result is sent in a client key exchange message. Since eavesdroppers do not know the server's private key, it will be infeasible for them to decode the pre_master_secret.

Note: No anonymous RSA Cipher Suites are defined in this document.

With Diffie-Hellman, the server's public parameters are contained in the server key exchange message and the client's are sent in the client key exchange message. Eavesdroppers who do not know the private values should not be able to find the Diffie-Hellman result (i.e. the pre_master_secret).

Warning: Completely anonymous connections only provide protection against passive eavesdropping. Unless an independent tamper-proof channel is used to verify that the finished messages were not replaced by an attacker, server authentication is required in environments where active man-in-the-middle attacks are a concern.

F.1.1.2. RSA key exchange and authentication

With RSA, key exchange and server authentication are combined. The public key may be either contained in the server's certificate or may be a temporary RSA key sent in a server key exchange message. When temporary RSA keys are used, they are signed by the server's RSA certificate. The signature includes the current ClientHello.random, so old signatures and temporary keys cannot be replayed. Servers may use a single temporary RSA key for multiple negotiation sessions.

Note: The temporary RSA key option is useful if servers need large certificates but must comply with government-imposed size limits on keys used for key exchange.

Note that if ephemeral RSA is not used, compromise of the server's static RSA key results in a loss of confidentiality for all sessions protected under that static key. TLS users desiring Perfect Forward Secrecy should use DHE cipher suites. The damage done by exposure of a private key can be limited by changing one's private key (and certificate) frequently.

After verifying the server's certificate, the client encrypts a pre_master_secret with the server's public key. By successfully decoding the pre_master_secret and producing a correct finished message, the server demonstrates that it knows the private key corresponding to the server certificate.

When RSA is used for key exchange, clients are authenticated using

the certificate verify message (see Section 7.4.10). The client signs a value derived from the master_secret and all preceding handshake messages. These handshake messages include the server certificate, which binds the signature to the server, and ServerHello.random, which binds the signature to the current handshake process.

F.1.1.3. Diffie-Hellman key exchange with authentication

When Diffie-Hellman key exchange is used, the server can either supply a certificate containing fixed Diffie-Hellman parameters or can use the server key exchange message to send a set of temporary Diffie-Hellman parameters signed with a DSS or RSA certificate. Temporary parameters are hashed with the hello.random values before signing to ensure that attackers do not replay old parameters. In either case, the client can verify the certificate or signature to ensure that the parameters belong to the server.

If the client has a certificate containing fixed Diffie-Hellman parameters, its certificate contains the information required to complete the key exchange. Note that in this case the client and server will generate the same Diffie-Hellman result (i.e., pre_master_secret) every time they communicate. To prevent the pre_master_secret from staying in memory any longer than necessary, it should be converted into the master_secret as soon as possible. Client Diffie-Hellman parameters must be compatible with those supplied by the server for the key exchange to work.

If the client has a standard DSS or RSA certificate or is unauthenticated, it sends a set of temporary parameters to the server in the client key exchange message, then optionally uses a certificate verify message to authenticate itself.

If the same DH keypair is to be used for multiple handshakes, either because the client or server has a certificate containing a fixed DH keypair or because the server is reusing DH keys, care must be taken to prevent small subgroup attacks. Implementations SHOULD follow the guidelines found in [SUBGROUP].

Small subgroup attacks are most easily avoided by using one of the DHE ciphersuites and generating a fresh DH private key (X) for each handshake. If a suitable base (such as 2) is chosen, g^X mod p can be computed very quickly so the performance cost is minimized. Additionally, using a fresh key for each handshake provides Perfect Forward Secrecy. Implementations SHOULD generate a new X for each handshake when using DHE ciphersuites.

F.1.2. Version rollback attacks

Because TLS includes substantial improvements over SSL Version 2.0, attackers may try to make TLS-capable clients and servers fall back to Version 2.0. This attack can occur if (and only if) two TLS-capable parties use an SSL 2.0 handshake.

Although the solution using non-random PKCS #1 block type 2 message padding is inelegant, it provides a reasonably secure way for Version 3.0 servers to detect the attack. This solution is not secure against attackers who can brute force the key and substitute a new ENCRYPTED-KEY-DATA message containing the same key (but with normal padding) before the application specified wait threshold has expired. Parties concerned about attacks of this scale should not be using 40-bit encryption keys anyway. Altering the padding of the least-significant 8 bytes of the PKCS padding does not impact security for the size of the signed hashes and RSA key lengths used in the protocol, since this is essentially equivalent to increasing the input block size by 8 bytes.

F.1.3. Detecting attacks against the handshake protocol

An attacker might try to influence the handshake exchange to make the parties select different encryption algorithms than they would normally chooses.

For this attack, an attacker must actively change one or more handshake messages. If this occurs, the client and server will compute different values for the handshake message hashes. As a result, the parties will not accept each others' finished messages. Without the master_secret, the attacker cannot repair the finished messages, so the attack will be discovered.

F.1.4. Resuming sessions

When a connection is established by resuming a session, new ClientHello.random and ServerHello.random values are hashed with the session's master_secret. Provided that the master_secret has not been compromised and that the secure hash operations used to produce the encryption keys and MAC secrets are secure, the connection should be secure and effectively independent from previous connections. Attackers cannot use known encryption keys or MAC secrets to compromise the master_secret without breaking the secure hash operations (which use both SHA and MD5).

Sessions cannot be resumed unless both the client and server agree. If either party suspects that the session may have been compromised, or that certificates may have expired or been revoked, it should force a full handshake. An upper limit of 24 hours is suggested for session ID lifetimes, since an attacker who obtains a master_secret

may be able to impersonate the compromised party until the corresponding session ID is retired. Applications that may be run in relatively insecure environments should not write session IDs to stable storage.

F.1.5 Extensions

Security considerations for the extension mechanism in general, and the design of new extensions, are described in the previous section. A security analysis of each of the extensions defined in this document is given below.

In general, implementers should continue to monitor the state of the art, and address any weaknesses identified.

F.1.5.1 Security of server_name

If a single server hosts several domains, then clearly it is necessary for the owners of each domain to ensure that this satisfies their security needs. Apart from this, server_name does not appear to introduce significant security issues.

Implementations MUST ensure that a buffer overflow does not occur whatever the values of the length fields in server_name.

Although this document specifies an encoding for internationalized hostnames in the server_name extension, it does not address any security issues associated with the use of internationalized hostnames in TLS - in particular, the consequences of "spoofed" names that are indistinguishable from another name when displayed or printed. It is recommended that server certificates not be issued for internationalized hostnames unless procedures are in place to mitigate the risk of spoofed hostnames.

6.2. Security of max_fragment_length

The maximum fragment length takes effect immediately, including for handshake messages. However, that does not introduce any security complications that are not already present in TLS, since [TLS] requires implementations to be able to handle fragmented handshake messages.

Note that as described in section XXX, once a non-null cipher suite has been activated, the effective maximum fragment length depends on the cipher suite and compression method, as well as on the negotiated max_fragment_length. This must be taken into account when sizing buffers, and checking for buffer overflow.

F.1.5.2 Security of client_certificate_url

There are two major issues with this extension.

The first major issue is whether or not clients should include certificate hashes when they send certificate URLs.

When client authentication is used *without* the client_certificate_url extension, the client certificate chain is covered by the Finished message hashes. The purpose of including hashes and checking them against the retrieved certificate chain, is to ensure that the same property holds when this extension is used - i.e., that all of the information in the certificate chain retrieved by the server is as the client intended.

On the other hand, omitting certificate hashes enables functionality that is desirable in some circumstances - for example clients can be issued daily certificates that are stored at a fixed URL and need not be provided to the client. Clients that choose to omit certificate hashes should be aware of the possibility of an attack in which the attacker obtains a valid certificate on the client's key that is different from the certificate the client intended to provide. Although TLS uses both MD5 and SHA-1 hashes in several other places, this was not believed to be necessary here. The property required of SHA-1 is second pre-image resistance.

The second major issue is that support for client_certificate_url involves the server acting as a client in another URL protocol. The server therefore becomes subject to many of the same security concerns that clients of the URL scheme are subject to, with the added concern that the client can attempt to prompt the server to connect to some, possibly weird-looking URL.

In general this issue means that an attacker might use the server to indirectly attack another host that is vulnerable to some security flaw. It also introduces the possibility of denial of service attacks in which an attacker makes many connections to the server, each of which results in the server attempting a connection to the target of the attack.

Note that the server may be behind a firewall or otherwise able to access hosts that would not be directly accessible from the public Internet; this could exacerbate the potential security and denial of service problems described above, as well as allowing the existence of internal hosts to be confirmed when they would otherwise be hidden.

The detailed security concerns involved will depend on the URL

schemes supported by the server. In the case of HTTP, the concerns are similar to those that apply to a publicly accessible HTTP proxy server. In the case of HTTPS, the possibility for loops and deadlocks to be created exists and should be addressed. In the case of FTP, attacks similar to FTP bounce attacks arise.

As a result of this issue, it is RECOMMENDED that the client_certificate_url extension should have to be specifically enabled by a server administrator, rather than being enabled by default. It is also RECOMMENDED that URI protocols be enabled by the administrator individually, and only a minimal set of protocols be enabled, with unusual protocols offering limited security or whose security is not well-understood being avoided.

As discussed in [URI], URLs that specify ports other than the default may cause problems, as may very long URLs (which are more likely to be useful in exploiting buffer overflow bugs).

Also note that HTTP caching proxies are common on the Internet, and some proxies do not check for the latest version of an object correctly. If a request using HTTP (or another caching protocol) goes through a misconfigured or otherwise broken proxy, the proxy may return an out-of-date response.

F.1.5.4. Security of trusted_ca_keys

It is possible that which CA root keys a client possesses could be regarded as confidential information. As a result, the CA root key indication extension should be used with care.

The use of the SHA-1 certificate hash alternative ensures that each certificate is specified unambiguously. As for the previous extension, it was not believed necessary to use both MD5 and SHA-1 hashes.

F.1.5.5. Security of truncated_hmac

It is possible that truncated MACs are weaker than "un-truncated" MACs. However, no significant weaknesses are currently known or expected to exist for HMAC with MD5 or SHA-1, truncated to 80 bits.

Note that the output length of a MAC need not be as long as the length of a symmetric cipher key, since forging of MAC values cannot be done off-line: in TLS, a single failed MAC guess will cause the immediate termination of the TLS session.

Since the MAC algorithm only takes effect after the handshake messages have been authenticated by the hashes in the Finished

messages, it is not possible for an active attacker to force negotiation of the truncated HMAC extension where it would not otherwise be used (to the extent that the handshake authentication is secure). Therefore, in the event that any security problem were found with truncated HMAC in future, if either the client or the server for a given session were updated to take into account the problem, they would be able to veto use of this extension.

F.1.5.6. Security of status_request

If a client requests an OCSP response, it must take into account that an attacker's server using a compromised key could (and probably would) pretend not to support the extension. A client that requires OCSP validation of certificates SHOULD either contact the OCSP server directly in this case, or abort the handshake.

Use of the OCSP nonce request extension (id-pkix-ocsp-nonce) may improve security against attacks that attempt to replay OCSP responses; see section 4.4.1 of [OCSP] for further details.

F.2. Protecting application data

The master_secret is hashed with the ClientHello.random and ServerHello.random to produce unique data encryption keys and MAC secrets for each connection.

Outgoing data is protected with a MAC before transmission. To prevent message replay or modification attacks, the MAC is computed from the MAC secret, the sequence number, the message length, the message contents, and two fixed character strings. The message type field is necessary to ensure that messages intended for one TLS Record Layer client are not redirected to another. The sequence number ensures that attempts to delete or reorder messages will be detected. Since sequence numbers are 64-bits long, they should never overflow. Messages from one party cannot be inserted into the other's output, since they use independent MAC secrets. Similarly, the server-write and client-write keys are independent so stream cipher keys are used only once.

If an attacker does break an encryption key, all messages encrypted with it can be read. Similarly, compromise of a MAC key can make message modification attacks possible. Because MACs are also encrypted, message-alteration attacks generally require breaking the encryption algorithm as well as the MAC.

Note: MAC secrets may be larger than encryption keys, so messages can
      remain tamper resistant even if encryption keys are broken.

F.3. Explicit IVs

[CBCATT] describes a chosen plaintext attack on TLS that depends on knowing the IV for a record. Previous versions of TLS [TLS1.0] used the CBC residue of the previous record as the IV and therefore enabled this attack. This version uses an explicit IV in order to protect against this attack.

F.4 Security of Composite Cipher Modes

TLS secures transmitted application data via the use of symmetric encryption and authentication functions defined in the negotiated ciphersuite. The objective is to protect both the integrity and confidentiality of the transmitted data from malicious actions by active attackers in the network. It turns out that the order in which encryption and authentication functions are applied to the data plays an important role for achieving this goal [ENCAUTH].

The most robust method, called encrypt-then-authenticate, first applies encryption to the data and then applies a MAC to the ciphertext. This method ensures that the integrity and confidentiality goals are obtained with ANY pair of encryption and MAC functions provided that the former is secure against chosen plaintext attacks and the MAC is secure against chosen-message attacks. TLS uses another method, called authenticate-then-encrypt, in which first a MAC is computed on the plaintext and then the concatenation of plaintext and MAC is encrypted. This method has been proven secure for CERTAIN combinations of encryption functions and MAC functions, but is not guaranteed to be secure in general. In particular, it has been shown that there exist perfectly secure encryption functions (secure even in the information theoretic sense) that combined with any secure MAC function fail to provide the confidentiality goal against an active attack. Therefore, new ciphersuites and operation modes adopted into TLS need to be analyzed under the authenticate-then-encrypt method to verify that they achieve the stated integrity and confidentiality goals.

Currently, the security of the authenticate-then-encrypt method has been proven for some important cases. One is the case of stream ciphers in which a computationally unpredictable pad of the length of the message plus the length of the MAC tag is produced using a pseudo-random generator and this pad is xor-ed with the concatenation of plaintext and MAC tag. The other is the case of CBC mode using a secure block cipher. In this case, security can be shown if one applies one CBC encryption pass to the concatenation of plaintext and MAC and uses a new, independent and unpredictable, IV for each new pair of plaintext and MAC. In previous versions of SSL, CBC mode was used properly EXCEPT that it used a predictable IV in the form of the last block of the previous ciphertext. This made TLS open to chosen plaintext attacks. This verson of the protocol is immune to those attacks. For exact details in the encryption modes proven secure see [ENCAUTH].

F.5 Denial of Service

TLS is susceptible to a number of denial of service (DoS) attacks. In particular, an attacker who initiates a large number of TCP connections can cause a server to consume large amounts of CPU doing RSA decryption. However, because TLS is generally used over TCP, it is difficult for the attacker to hide his point of origin if proper TCP SYN randomization is used [SEQNUM] by the TCP stack.

Because TLS runs over TCP, it is also susceptible to a number of denial of service attacks on individual connections. In particular, attackers can forge RSTs, terminating connections, or forge partial TLS records, causing the connection to stall. These attacks cannot in general be defended against by a TCP-using protocol. Implementors or users who are concerned with this class of attack should use IPsec AH [AH] or ESP [ESP].

F.6. Final notes

For TLS to be able to provide a secure connection, both the client and server systems, keys, and applications must be secure. In addition, the implementation must be free of security errors.

The system is only as strong as the weakest key exchange and authentication algorithm supported, and only trustworthy cryptographic functions should be used. Short public keys, 40-bit bulk encryption keys, and anonymous servers should be used with great caution. Implementations and users must be careful when deciding which certificates and certificate authorities are acceptable; a dishonest certificate authority can do tremendous damage.

Security Considerations

Security issues are discussed throughout this memo, especially in Appendices D, E, and F.

Normative References

[AES] National Institute of Standards and Technology, "Specification for the Advanced Encryption Standard (AES)" FIPS 197. November 26, 2001.

[3DES] W. Tuchman, "Hellman Presents No Shortcut Solutions To DES," IEEE Spectrum, v. 16, n. 7, July 1979, pp40-41.

[DES] ANSI X3.106, "American National Standard for Information Systems-Data Link Encryption," American National Standards Institute, 1983.

[DSS] NIST FIPS PUB 186-2, "Digital Signature Standard," National Institute of Standards and Technology, U.S. Department of Commerce, 2000.

[HMAC] Krawczyk, H., Bellare, M., and R. Canetti, "HMAC: Keyed-Hashing for Message Authentication," RFC 2104, February 1997.

[HTTP] Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P. and T. Berners-Lee, "Hypertext Transfer Protocol -- HTTP/1.1", RFC 2616, June 1999.

[IDEA] X. Lai, "On the Design and Security of Block Ciphers," ETH Series in Information Processing, v. 1, Konstanz: Hartung-Gorre Verlag, 1992.

[IDNA] Faltstrom, P., Hoffman, P. and A. Costello, "Internationalizing Domain Names in Applications (IDNA)", RFC 3490, March 2003.

[MD5] Rivest, R., "The MD5 Message Digest Algorithm", RFC 1321, April 1992.

[OCSP] Myers, M., Ankney, R., Malpani, A., Galperin, S. and C. Adams, "Internet X.509 Public Key Infrastructure: Online Certificate Status Protocol - OCSP", RFC 2560, June 1999.

[PKCS1A] B. Kaliski, "Public-Key Cryptography Standards (PKCS) #1: RSA Cryptography Specifications Version 1.5", RFC 2313, March 1998.

[PKCS1B] J. Jonsson, B. Kaliski, "Public-Key Cryptography Standards (PKCS) #1: RSA Cryptography Specifications Version 2.1", RFC 3447, February 2003.

[PKIOP] Housley, R. and P. Hoffman, "Internet X.509 Public Key Infrastructure - Operation Protocols: FTP and HTTP", RFC 2585, May 1999.

[PKIX] Housley, R., Ford, W., Polk, W. and D. Solo, "Internet Public Key Infrastructure: Part I: X.509 Certificate and CRL Profile", RFC 3280, April 2002.

[RC2] Rivest, R., "A Description of the RC2(r) Encryption Algorithm", RFC 2268, January 1998.

[SCH] B. Schneier. "Applied Cryptography: Protocols, Algorithms, and Source Code in C, 2ed", Published by John Wiley & Sons, Inc. 1996.

[SHA] NIST FIPS PUB 180-2, "Secure Hash Standard," National Institute of Standards and Technology, U.S. Department of Commerce., August 2001.

[REQ] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

[RFC2434] T. Narten, H. Alvestrand, "Guidelines for Writing an IANA Considerations Section in RFCs", RFC 3434, October 1998.

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

[TLSEXT] Blake-Wilson, S., Nystrom, M, Hopwood, D., Mikkelsen, J., Wright, T., "Transport Layer Security (TLS) Extensions", RFC 3546, June 2003.

[TLSKRB] A. Medvinsky, M. Hur, "Addition of Kerberos Cipher Suites to Transport Layer Security (TLS)", RFC 2712, October 1999.

[URI] Berners-Lee, T., Fielding, R. and L. Masinter, "Uniform

Resource Identifiers (URI): Generic Syntax", RFC 2396, August 1998.

[UTF8] Yergeau, F., "UTF-8, a transformation format of ISO 10646", RFC 3629, November 2003.

[X509-4th] ITU-T Recommendation X.509 (2000) | ISO/IEC 9594- 8:2001,

"Information Systems - Open Systems Interconnection - The Directory: Public key and Attribute certificate frameworks."

[X509-4th-TC1] ITU-T Recommendation X.509(2000) Corrigendum 1(2001) | ISO/IEC 9594-8:2001/Cor.1:2002, Technical Corrigendum 1 to ISO/IEC 9594:8:2001.

Informative References

[AH] Kent, S., and Atkinson, R., "IP Authentication Header", RFC 2402, November 1998.

[BLEI] Bleichenbacher D., "Chosen Ciphertext Attacks against Protocols Based on RSA Encryption Standard PKCS #1" in Advances in Cryptology -- CRYPTO'98, LNCS vol. 1462, pages: 1-12, 1998.

[CBCATT] Moeller, B., "Security of CBC Ciphersuites in SSL/TLS: Problems and Countermeasures", http://www.openssl.org/~bodo/tls-cbc.txt.

[CBCTIME] Canvel, B., "Password Interception in a SSL/TLS Channel", http://lasecwww.epfl.ch/memo_ssl.shtml, 2003.

[ENCAUTH] Krawczyk, H., "The Order of Encryption and Authentication for Protecting Communications (Or: How Secure is SSL?)", Crypto 2001.

[ESP] Kent, S., and Atkinson, R., "IP Encapsulating Security Payload (ESP)", RFC 2406, November 1998.

[KPR03] Klima, V., Pokorny, O., Rosa, T., "Attacking RSA-based Sessions in SSL/TLS", http://eprint.iacr.org/2003/052/, March 2003.

[PKCS6] RSA Laboratories, "PKCS #6: RSA Extended Certificate Syntax Standard," version 1.5, November 1993.

[PKCS7] RSA Laboratories, "PKCS #7: RSA Cryptographic Message Syntax Standard," version 1.5, November 1993.

[RANDOM] D. Eastlake 3rd, S. Crocker, J. Schiller. "Randomness Recommendations for Security", RFC 1750, December 1994.

[RSA]   R. Rivest, A. Shamir, and L. M. Adleman, "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems," Communications of the ACM, v. 21, n. 2, Feb 1978, pp.

120-126.

[SEQNUM] Bellovin. S., "Defending Against Sequence Number Attacks", RFC 1948, May 1996.

[SSL2]   Hickman, Kipp, "The SSL Protocol", Netscape Communications Corp., Feb 9, 1995.

[SSL3]   A. Frier, P. Karlton, and P. Kocher, "The SSL 3.0 Protocol", Netscape Communications Corp., Nov 18, 1996.

[SUBGROUP] R. Zuccherato, "Methods for Avoiding the Small-Subgroup Attacks on the Diffie-Hellman Key Agreement Method for S/MIME", RFC 2785, March 2000.

[TCP]   Postel, J., "Transmission Control Protocol," STD 7, RFC 793, September 1981.

[TIMING] Boneh, D., Brumley, D., "Remote timing attacks are practical", USENIX Security Symposium 2003.

[TLS1.0] Dierks, T., and Allen, C., "The TLS Protocol, Version 1.0", RFC 2246, January 1999.

[TLS1.1] Dierks, T., and Rescorla, E., "The TLS Protocol, Version 1.1", RFC 4346, April, 2006.

[X501] ITU-T Recommendation X.501: Information Technology - Open Systems Interconnection - The Directory: Models, 1993.

[X509] ITU-T Recommendation X.509 (1997 E): Information Technology - Open Systems Interconnection - "The Directory - Authentication Framework". 1988.

[XDR]   R. Srinivansan, Sun Microsystems, "XDR: External Data

         Representation Standard", RFC 1832, August 1995.

Credits

Working Group Chairs
Eric Rescorla
EMail: ekr@rtfm.com

Pasi Eronen
pasi.eronen@nokia.com

Editors

```
Tim Dierks                  Eric Rescorla
Independent                     Network Resonance, Inc.

EMail: tim@dierks.org           EMail: ekr@networkresonance.com
```

Other contributors

Christopher Allen (co-editor of TLS 1.0)
Alacrity Ventures
ChristopherA@AlacrityManagement.com

Martin Abadi
University of California, Santa Cruz
abadi@cs.ucsc.edu

Steven M. Bellovin
Columbia University
smb@cs.columbia.edu

Simon Blake-Wilson
BCI
EMail: sblakewilson@bcisse.com

Ran Canetti
IBM
canetti@watson.ibm.com

Pete Chown

Skygate Technology Ltd
pc@skygate.co.uk

Taher Elgamal
taher@securify.com
Securify

Anil Gangolli
anil@busybuddha.org

Kipp Hickman

David Hopwood
Independent Consultant
EMail: david.hopwood@blueyonder.co.uk

Phil Karlton (co-author of SSLv3)

Paul Kocher (co-author of SSLv3)
Cryptography Research
paul@cryptography.com

Hugo Krawczyk
Technion Israel Institute of Technology
hugo@ee.technion.ac.il

Jan Mikkelsen
Transactionware
EMail: janm@transactionware.com

Magnus Nystrom
RSA Security
EMail: magnus@rsasecurity.com

Robert Relyea
Netscape Communications
relyea@netscape.com

Jim Roskind
Netscape Communications
jar@netscape.com

Michael Sabin

Dan Simon

Microsoft, Inc.
dansimon@microsoft.com

Tom Weinstein

Tim Wright
Vodafone
EMail: timothy.wright@vodafone.com

Comments

The discussion list for the IETF TLS working group is located at the e-mail address <tls@ietf.org>. Information on the group and information on how to subscribe to the list is at <https://www1.ietf.org/mailman/listinfo/tls>

Archives of the list can be found at:
<http://www.ietf.org/mail-archive/web/tls/current/index.html>

Copyright Statement

Acknowledgment

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