Network Working Group
Request for Comments: 2246
Category: Standards Track

T. Dierks
Certicom
C. Allen
Certicom
January 1999

The TLS Protocol
Version 1.0

Status of this Memo

This document specifies an Internet standards track protocol for the Internet community, and requests discussion and suggestions for improvements. Please refer to the current edition of the "Internet Official Protocol Standards" (STD 1) for the standardization state and status of this protocol. Distribution of this memo is unlimited.

Copyright Notice

Copyright (C) The Internet Society (1999). All Rights Reserved.

Abstract

This document specifies Version 1.0 of the Transport Layer Security (TLS) protocol. The TLS protocol provides communications privacy 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

1. Introduction 3
2. Goals 4
3. Goals of this document 5
4. Presentation language 5
4.1. Basic block size 6
4.2. Miscellaneous 6
4.3. Vectors 6
4.4. Numbers 7
4.5. Enumerateds 7
4.6. Constructed types 8
4.6.1. Variants 9
4.7. Cryptographic attributes 10
4.8. Constants 11
5. HMAC and the pseudorandom function 11
6. The TLS Record Protocol 13
6.1. Connection states 14

AMAZON-1028
IPR2026-00088, IPR2026-00106

6.2. Record layer 16
6.2.1. Fragmentation 16
6.2.2. Record compression and decompression 17
6.2.3. Record payload protection 18
6.2.3.1. Null or standard stream cipher 19
6.2.3.2. CBC block cipher 19
6.3. Key calculation 21
6.3.1. Export key generation example 22
7. The TLS Handshake Protocol 23
7.1. Change cipher spec protocol 24
7.2. Alert protocol 24
7.2.1. Closure alerts 25
7.2.2. Error alerts 26
7.3. Handshake Protocol overview 29
7.4. Handshake protocol 32
7.4.1. Hello messages 33
7.4.1.1. Hello request 33
7.4.1.2. Client hello 34
7.4.1.3. Server hello 36
7.4.2. Server certificate 37
7.4.3. Server key exchange message 39
7.4.4. Certificate request 41
7.4.5. Server hello done 42
7.4.6. Client certificate 43
7.4.7. Client key exchange message 43
7.4.7.1. RSA encrypted premaster secret message 44
7.4.7.2. Client Diffie-Hellman public value 45
7.4.8. Certificate verify 45
7.4.9. Finished 46
8. Cryptographic computations 47
8.1. Computing the master secret 47
8.1.1. RSA 48
8.1.2. Diffie-Hellman 48
9. Mandatory Cipher Suites 48
10. Application data protocol 48
A. Protocol constant values 49
A.1. Record layer 49
A.2. Change cipher specs message 50
A.3. Alert messages 50
A.4. Handshake protocol 51
A.4.1. Hello messages 51
A.4.2. Server authentication and key exchange messages 52
A.4.3. Client authentication and key exchange messages 53
A.4.4. Handshake finalization message 54
A.5. The CipherSuite 54
A.6. The Security Parameters 56
B. Glossary 57
C. CipherSuite definitions 61

D. Implementation Notes 64
D.1. Temporary RSA keys 64
D.2. Random Number Generation and Seeding 64
D.3. Certificates and authentication 65
D.4. CipherSuites 65
E. Backward Compatibility With SSL 66
E.1. Version 2 client hello 67
E.2. Avoiding man-in-the-middle version rollback 68
F. Security analysis 69
F.1. Handshake protocol 69
F.1.1. Authentication and key exchange 69
F.1.1.1. Anonymous key exchange 69
F.1.1.2. RSA key exchange and authentication 70
F.1.1.3. Diffie-Hellman key exchange with authentication 71
F.1.2. Version rollback attacks 71
F.1.3. Detecting attacks against the handshake protocol 72
F.1.4. Resuming sessions 72
F.1.5. MD5 and SHA 72
F.2. Protecting application data 72
F.3. Final notes 73
G. Patent Statement 74
Security Considerations 75
References 75
Credits 77
Comments 78
Full Copyright Statement 80

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 [RC4], 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.

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 TLS 1.0 and SSL 3.0 do not interoperate (although TLS 1.0 does incorporate a mechanism by which a TLS implementation can back down to SSL 3.0). 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, a 36-byte structure of two hashes (one SHA and one MD5) is signed (encrypted with the private key). It is encoded with PKCS #1 block type 0 or type 1 as described in [PKCS1].

In DSS, the 20 bytes of the SHA 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 [PKCS1].

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].

HMAC can be used with a variety of different hash algorithms. TLS uses it in the handshake with two different algorithms: MD5 and SHA-1, denoting these as HMAC_MD5(secret, data) and HMAC_SHA(secret,

data). Additional hash algorithms can be defined by cipher suites and used to protect record data, but MD5 and SHA-1 are hard coded into the description of the handshaking for this version of the protocol.

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.

In order to make the PRF as secure as possible, it uses two hash algorithms in a way which should guarantee its security if either algorithm remains secure.

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 splitting the secret into two halves and using one half to generate data with P_MD5 and the other half to generate data with P_SHA-1, then exclusive-or'ing the outputs of these two expansion functions together.

S1 and S2 are the two halves of the secret and each is the same length. S1 is taken from the first half of the secret, S2 from the second half. Their length is created by rounding up the length of the overall secret divided by two; thus, if the original secret is an odd number of bytes long, the last byte of S1 will be the same as the first byte of S2.

```
L_S = length in bytes of secret;
L_S1 = L_S2 = ceil(L_S / 2);
```

The secret is partitioned into two halves (with the possibility of one shared byte) as described above, S1 taking the first L_S1 bytes and S2 the last L_S2 bytes.

The PRF is then defined as the result of mixing the two pseudorandom streams by exclusive-or'ing them together.

```
PRF(secret, label, seed) = P_MD5(S1, label + seed) XOR
                           P_SHA-1(S2, label + seed);
```

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
```

Note that because MD5 produces 16 byte outputs and SHA-1 produces 20 byte outputs, the boundaries of their internal iterations will not be aligned; to generate a 80 byte output will involve P_MD5 being iterated through A(5), while P_SHA-1 will only iterate through A(4).

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.2). 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 take 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 and IVs 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 Handshake Protocol 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), and whether it is considered an "export" cipher.

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 } BulkCipherAlgorithm;

enum { stream, block } CipherType;

enum { true, false } IsExportable;

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;
    IsExportable           is_exportable;
    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 six items:

```
client write MAC secret
server write MAC secret
client write key
server write key
client write IV (for block ciphers only)
server write IV (for block ciphers only)
```

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. In addition, for block ciphers running in CBC mode (the only mode specified for TLS), this will initially contain the IV for that connection state and be updated to contain the ciphertext of the last block encrypted or decrypted as records are processed. For stream ciphers, this will 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. A sequence number is incremented after each record: specifically, the first record which is transmitted under a particular connection state should 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 1.0, which uses the version { 3, 1 }. The version value 3.1 is historical: TLS version 1.0 is a minor modification to the SSL 3.0 protocol, which bears the version value 3.0. (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.

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 or DES), the encryption and MAC functions convert TLSCompressed.fragment structures to and from block TLSCiphertext.fragment structures.

```
block-ciphered struct {
    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.

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.

padding_length
: The padding length should 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. 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) the initialization vector (IV) for the first record is generated with the other keys and secrets when the security parameters are set. The IV for subsequent records is the last ciphertext block from the previous record.

6.3. Key calculation

The Record Protocol requires an algorithm to generate keys, IVs, 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, keys, and non-export IVs 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, a server write key, a client write IV, and a server write IV, 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, and the random values provide unencrypted salt material and IVs for exportable ciphers.

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]
client_write_IV[SecurityParameters.IV_size]
server_write_IV[SecurityParameters.IV_size]
```

The client_write_IV and server_write_IV are only generated for non-export block ciphers. For exportable block ciphers, the initialization vectors are generated later, as described below. Any extra key_block material is discarded.

Implementation note:
The cipher spec which is defined in this document which requires the most material is 3DES_EDE_CBC_SHA: it requires 2 x 24 byte keys, 2 x 20 byte MAC secrets, and 2 x 8 byte IVs, for a total of 104 bytes of key material.

Exportable encryption algorithms (for which CipherSpec.is_exportable is true) require additional processing as follows to derive their final write keys:

```
final_client_write_key =
PRF(SecurityParameters.client_write_key,
                           "client write key",
                           SecurityParameters.client_random +
                           SecurityParameters.server_random);
final_server_write_key =
PRF(SecurityParameters.server_write_key,
                           "server write key",
                           SecurityParameters.client_random +
                           SecurityParameters.server_random);
```

Exportable encryption algorithms derive their IVs solely from the random values from the hello messages:

```
iv_block = PRF("", "IV block", SecurityParameters.client_random +
               SecurityParameters.server_random);
```

The iv_block is partitioned into two initialization vectors as the key_block was above:

```
client_write_IV[SecurityParameters.IV_size]
server_write_IV[SecurityParameters.IV_size]
```

Note that the PRF is used without a secret in this case: this just means that the secret has a length of zero bytes and contributes nothing to the hashing in the PRF.

6.3.1. Export key generation example

TLS_RSA_EXPORT_WITH_RC2_CBC_40_MD5 requires five random bytes for each of the two encryption keys and 16 bytes for each of the MAC keys, for a total of 42 bytes of key material. The PRF output is stored in the key_block. The key_block is partitioned, and the write keys are salted because this is an exportable encryption algorithm.

```
key_block               = PRF(master_secret,
                              "key expansion",
                              server_random +
                              client_random)[0..41]
client_write_MAC_secret = key_block[0..15]
server_write_MAC_secret = key_block[16..31]
client_write_key        = key_block[32..36]
server_write_key        = key_block[37..41]
```

```
final_client_write_key  = PRF(client_write_key,
                              "client write key",
                              client_random +
                              server_random)[0..15]
final_server_write_key  = PRF(server_write_key,
                              "server write key",
                              client_random +
                              server_random)[0..15]

iv_block                = PRF("", "IV block", client_random +
                              server_random)[0..15]
client_write_IV = iv_block[0..7]
server_write_IV = iv_block[8..15]
```

7. The TLS Handshake Protocol

The TLS Handshake Protocol consists of a suite of three sub-protocols 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 should 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 7.4.9).

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),
            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(60),
            protocol_version(70),
            insufficient_security(71),
            internal_error(80),
            user_canceled(90),
            no_renegotiation(100),
            (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. The session becomes unresumable if any connection is terminated without proper close_notify messages with level equal to warning.

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

Each party is required to send a close_notify alert before closing the write side of the connection. It is required that the other party 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.

NB: 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 are required to forget any session-identifiers, keys, and secrets associated with a failed connection. 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 message is always fatal.

decryption_failed
: 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.

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.

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
A negotiation not in compliance with export restrictions was detected; for example, attempting to transfer a 1024 bit ephemeral RSA key for the RSA_EXPORT handshake method. This message is always fatal.

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.

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.

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 should 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.)

```
Client                                               Server

ClientHello                  -------->
                                                ServerHello
                                               Certificate*
                                         ServerKeyExchange*
                                        CertificateRequest*
                             <--------      ServerHelloDone
Certificate*
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), (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;
    } 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 in 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.

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 should never 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) 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;
```

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 3.1 (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.

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.

Forward compatibility note:
In the interests of forward compatibility, it is permitted for a client hello message to include extra data after the compression methods. This data must be included in the handshake hashes, but must otherwise be ignored. This is the only handshake message for which this is legal; for all other messages, the amount of data in the message must match the description of the message precisely.

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;
```

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 3.1 (See Appendix E for details about backward compatibility).

random
This structure is generated by the server and must be different from (and independent of) 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.

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.
RSA_EXPORT	RSA public key of length greater than 512 bits which can be used for signing, or a key of 512 bits or shorter which can be used for either encryption or signing.
DHE_DSS	DSS public key.
DHE_DSS_EXPORT	DSS public key.
DHE_RSA	RSA public key which can be used for signing.
DHE_RSA_EXPORT	RSA public key which can be used for signing.
DH_DSS	Diffie-Hellman key. The algorithm used to sign the certificate should be DSS.
DH_RSA	Diffie-Hellman key. The algorithm used to sign the certificate should 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:

RSA_EXPORT (if the public key in the server certificate is longer than 512 bits)
DHE_DSS
DHE_DSS_EXPORT
DHE_RSA
DHE_RSA_EXPORT
DH_anon

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

RSA
RSA_EXPORT (when the public key in the server certificate is less than or equal to 512 bits in length)
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.

Note: According to current US export law, RSA moduli larger than 512 bits may not be used for key exchange in software exported from the US. With this message, the larger RSA keys encoded in certificates may be used to sign temporary shorter RSA keys for the RSA_EXPORT key exchange method.

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;

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.

md5_hash
    MD5(ClientHello.random + ServerHello.random + ServerParams);

sha_hash
    SHA(ClientHello.random + ServerHello.random + ServerParams);

enum { anonymous, rsa, dsa } SignatureAlgorithm;

select (SignatureAlgorithm)
{   case anonymous: struct { };
    case rsa:
        digitally-signed struct {
            opaque md5_hash[16];
            opaque sha_hash[20];
        };
    case dsa:
        digitally-signed struct {
            opaque sha_hash[20];
        };
} Signature;
```

7.4.4. 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),
    (255)
} ClientCertificateType;

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

struct {
    ClientCertificateType certificate_types<1..2^8-1>;
    DistinguishedName certificate_authorities<3..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_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.

Note: DistinguishedName is derived from [X509].

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

7.4.5. 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;
```

7.4.6. 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. 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.

7.4.7. Client key exchange message

When this message will be sent:
This message is always sent by the client. It will immediately follow the client certificate message, if it is sent. Otherwise it will 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 will 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;
```

7.4.7.1. 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;
```

Note: An attack discovered by Daniel Bleichenbacher [BLEI] can be used to attack a TLS server which is using PKCS#1 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 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.