Microsoft Strawman Proposal for a Secure Transport Layer Protocol ("STLP")

Discussion Draft 1.0
April 8, 1996

This document is intended to provide a first basis for discussion in the Transport Layer Security Working Group of the IETF. The stated charter of this group is "to write standards track RFC(s) for protocols using the currently available Internet drafts as a basis. The SSL, PCT and SSH protocols are examples of mechanisms of establishing a secure channel for general purpose or special purpose Internet applications running over a reliable transport, usually TCP."

To expedite this process, Microsoft and Netscape have agreed to start face-to-face discussions with the objective of presenting a viable draft proposal to the IETF working group at its next meeting in June.

Acknowledging that SSL is the more widely implemented protocol at this time, this discussion draft assumes it as a base protocol. This does not imply, however, that Microsoft (or the working group) either endorses SSLv3 "as is" or expects that structural changes to the base protocol beyond those presented in this document will not evolve as part of this open process.

Comments are solicited and may be posted to either ietf-tls@w3.org or to a new list specifically for this discussion at the discretion of the IETF TLS Working Group Chair, Win Treese <treese@OpenMarket.com>. Notification of this list will be made by April 10.

Contents

SSLv3 Implementation Comments

The following non-comprehensive list is derived from implementation experience with SSL Version 3 published as an Internet Draft in March 1996. Its intent is to suggest modifications to the base protocol rather than to specifically critique SSL Version 3.

(1) There is no significant implementation cost for separate negotiation of the message digest algorithm and the bulk cypher. In fact, the code could be better if they were separate; instead of one large table with a lot of repetition between entries there would be two small tables with no repetition.

(2) Don’t use post-compressed, post-hashed, post-encrypted, post-enveloped data for the final handshake digest. Implementation is painful. Using the original cleartext, or alternatively, the compressedtext, makes for a cleaner layer structure in the implementation.

(3) Provide a stronger error reportage. If a packet is refused, there should be notification. The hardest thing in implementing SSL is figuring out why some anonymous server refused your record. Also, it is desirable to be able to explain to the user what the problem is.

(4) Provision for a uniform message header for all types of messages is a very good thing. At any time in the process it should be possible to differentiate control from client data.

(5) Short-circuited handshaking is a very good thing. The session id stuff helps but doesn't go far enough. It is highly desirable that downloading a web page and five graphics on it not require so much handshaking.

(6) Password authentication (particularly for clients) is extremely desirable. Right now, it has to be done at an application protocol level (and differently for every protocol). Having part of authentication occur at the SSL level and part at the application protocol level is not desirable.

(7) Greater symmetry would be valuable from an implementation point of view. If a given piece of data is being exchanged, it should be the same whether the server or the client is supplying it (there should not be two different message formats for supplying a certificate). Right now, implementing the server and implementing the client are almost unrelated tasks because the protocol is so asymmetrical.

(8) MD2 and MD4 need to be phased out owing to the detection of a security problem. SHA is recommended.

(9) There is real value in a secure datagram, particularly for broadcast and multicast purposes.

(10) It is highly desirable that record formats provide for aligned data (i.e. 2 byte items on two byte boundaries, 4 byte plus items on 4 byte boundaries) if at all possible. Not all compilers can close-pack structures, forcing the implementer to roll and un-roll the records in his own code. This makes the implementation more expensive, error-prone, slower, and less readable. An extra byte here and there in a record is a small price to pay.

The STLP Protocol: A proposed merger of SSL 3.0 and PCT 2

April 1996

Caution: Extremely rough, preliminary draft—<u>not</u> for wide distribution

1. Introduction

This document gives a very rough, extremely preliminary outline of a secure transport layer protocol, STLP, drawing on two other protocols, SSL version 3.0 and PCT version 2. It is based primarily on the former, and retains most of its form and formalisms (including large chunks of borrowed text), but incorporates a number of features of PCT version 2.0. It is meant as a working document for a proposed merger of the two protocols, in the hope that eventually a single protocol can be accepted as an agreed-upon standard for channel security over the Internet.

2. Changes from SSL version 3.0

The changes made to SSL version 3.0 to produce STLP can be summarized as follows:

- A new `application_datagram` message type has been added, for transmission of independently decryptable "datagrams". This message includes a new `KeyInfoBlock` field, which contains the information necessary to decrypt the datagram. Fields have also been added to the CipherSpec data block to make key per-record key derivation possible.

- The **change cipher spec** protocol has been altered to allow the transmission of a new key and cipher specification. This feature makes possible the pre-encryption of data for later transmission. MAC calculation has also been altered for this purpose.

- MAC calculation and key derivation have been made more uniform, and changed from use of fixed, specified hash functions to a single hash function chosen during the handshake, to avoid long-term dependence on the security properties of any fixed algorithms.

- Alert messages indicating a failure to negotiate an encryption or certificate attribute due to a mismatch in supported attributes have been given an added informational field specifying the exact source of the mismatch.

- The handshake protocol has been reorganized to be more symmetric; for example, a uniform **hello** message is defined, which can be used by either client or server to initiate the protocol. Sender-independent **public value** and **response** messages are also defined, replacing sender-specific ones. The resulting handshake is much more flexible, allowing, for example, "quick" key exchanges by clients who already possess the server's public key exchange key. The uniform **response** message format makes the verifying hash in the **finished** message unnecessary, and this now-empty message has been converted to use as a delimiter for groups of handshake messages.

- Authentication options are explicitly negotiated; a new “password” authentication method is added, to allow either client or server to authenticate by previously shared private password.
- Full specification of certificate types and certifier names is extended to both client and server certificates.
- UNIX time is removed from the random challenges, to preserve sources of randomness.

3. Presentation language

The presentation language is taken directly from the one used for SSL version 3.0.

3.1 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.

3.2 Miscellaneous

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

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

Single byte entities containing uninterpreted data are of type opaque.

3.3 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 of data */
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 3.4). 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.

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

3.4 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 3.1 and are also unsigned. The following numeric types are predefined.

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

3.5 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 (v1), ... , 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, but legal */
Color color = blue;     /* correct, type is implicit */
```

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

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

3.6 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.

3.6.1 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 variant
                    selector is implicit */
            case apple: V1;        /* definition of
                    VariantBody, tag = apple */
            case orange: V2;       /* definition of
                    VariantBody, tag = orange */
        } variant_body;        /* optional label on the
                    variant portion */
} 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`.

3.7 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 4.1).

In digital signing, the data to be signed is first input into a collision-intractable one-way hash function, with the result then transformed using a signature private key to produce a digital signature for the input data.

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. Because it is unlikely that the plaintext (whatever data is to be sent) will break neatly into the necessary block size (usually 64 bits), it is necessary to pad out the end of short blocks with some regular pattern, usually all zeroes.

In public key encryption, one-way functions with secret "trapdoors" are used to encrypt the outgoing data. Data encrypted with the public key of a given key pair can only be decrypted with the private key.

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 a signing algorithm, then the entire structure is encrypted with a stream cipher. Note that the digital signature operation includes application of a collision-intractable one-way hash function the input; if `hash` is the output of a collision-intractable one-way hash function, then the result is once again passed through such a hash function as part of the digital signature operation.

6.8 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 */
```

4. The STLP protocol

STLP is a layered protocol. At each layer, messages may include fields for length, description, and content. STLP 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.

4.1 Session and connection states

A STLP session is *stateful*. It is the responsibility of the STLP Handshake protocol to coordinate the states of the client and server, thereby allowing the protocol state machines of each to operate consistently, despite the fact that the state is not exactly parallel. Logically the state is represented twice, once as the current *operating* state, and (during the handshake protocol) again as the *pending* state. Additionally, separate read and write states are maintained. When the client or server receives a **change cipher spec** message, it copies the pending read state (including information obtained from the **change cipher spec** message itself) into the current read state. When the client or server sends a **change cipher spec** message, it copies the pending write state into the current write state. When the handshake negotiation is complete, the client and server exchange **change cipher spec** messages (see Section 4.3), and then communicate using the newly agreed-upon specifications. The **change cipher spec** message may also, in certain cases, contain enough information for a pending write state to be constructed immediately to replace the current one; see section 4.3.

A STLP session may include multiple secure connections; in addition, parties may have multiple simultaneous sessions.

The session state includes the following elements:

session identifier
An arbitrary byte sequence chosen by the server to identify an active or resumable session state
peer certificate
The certificate of the other party to the session. This element of the state may be `NULL`. (A certificate is a digitally signed data block linking an asymmetric public key to a particular identity.

The STLP protocol assumes that parties indicating support for a particular certificate type, or from a particular certifying authority, possess the means to assess the validity of certificates of that type or from that certifier. Those means might include not only cryptographic operations, such as digital signature verification, but also such actions as comparison with certificate revocation lists, examination of expiry dates, or even presentation to a user for approval.)

compression method

The algorithm used to compress data prior to encryption**.**

CipherSpec

Specifies the bulk data encryption algorithm (such as `NULL`, `DES`, etc.) and a cryptographic hashing algorithm (such as MD5 or SHA) for message authentication code (MAC) computation. It also defines cryptographic attributes such as the `hash_size`. (See Appendix A for a formal definition.)

Alternate Specs

A list of alternate cipher, hash and key exchange algorithms for which both parties have indicated support. This list can be used to determine if pre-encrypted data will be decryptable if sent.

master secret

48-byte secret shared between the client and server. This value is included in the CipherSpec.

is resumable

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

The connection state includes the following elements:

server and client random

Byte sequences that are chosen by the server and client for each connection.

sequence numbers

Each party maintains separate sequence numbers for transmitted and received messages for each connection. When a party sends or receives a **change cipher spec** message, the appropriate sequence number is set to zero. Sequence numbers are of type `uint32` and may not exceed 2^{32}-1. A CipherSpec counter (also of type `uint32`) is also kept for transmitted and received messages; this counter is incremented each time a sequence of pre-encrypted data records is sent.

Note: These random values and sequence numbers are grouped together into the `ConnectionParams` structure; see Appendix A.

server write MAC secret

The secret used in MAC operations on data written by the server.

client write MAC secret

The secret used in MAC operations on data written by the client.

server write key

The bulk cipher key for data encrypted by the server and decrypted by the client.

client write key

The bulk cipher key for data encrypted by the client and decrypted by the server.

initialization vectors

When a block cipher in CBC mode is used, an initialization vector (IV) is maintained for each key. This field is first initialized by the STLP handshake protocol. Thereafter the final ciphertext block from each record is preserved for use with the following record.

4.2 Record layer

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

4.2.1 Fragmentation

The record layer fragments information blocks into STLPPlaintext records 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 STLPPlaintext record).

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

enum {
     change_cipher_spec(20), alert(21), handshake(22),
application_data(23), application_datagram(24), (255)
} ContentType;

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

`type`
The higher level protocol used to process the enclosed fragment.
`version`
The version of protocol being employed. This document describes STLP Version 1.0.
`length`
The length (in bytes) of the following STLPPlaintext.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 STLP Record layer content types may be interleaved. Application data is generally of lower precedence for transmission than other content types.

4.2.2 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 `NULL`.

The compression algorithm translates a STLPPlaintext structure into a STLPCompressed structure.
Compression functions erase their state information whenever the CipherSpec is replaced.

Note: The CipherSpec is part of the session state described in Section 4.1. References to fields of the CipherSpec are made throughout this document using presentation syntax. A more complete description of the CipherSpec is shown in Appendix A.

Compression must be lossless and may not increase the content length by more than 1024 bytes. If the decompression function encounters a `STLPCompressed.fragment` that would decompress to a length in excess of 2^{14} bytes, it should issue a fatal `decompression_failure` alert (see Section 4.4.2).

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

`type, version`
Identical to the values `STLPPlaintext.type and .version.`
`length`
The length (in bytes) of the following `STLPCompressed.fragment`. The length should not exceed 2^{14} + 1024.
`fragment`
The compressed form of `STLPPlaintext.fragment`.

Note: `A 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.

4.2.3 Record payload protection and the CipherSpec

All records are protected using the encryption and MAC algorithms defined in the current CipherSpec. There is always an active CipherSpec; however, its initial values specify null encryption and MAC algorithms, and hence do not provide any security.

Once the handshake is complete, the two parties have shared secrets which are used to encrypt records and compute keyed message authentication codes (MACs) on their contents. The techniques used to perform the encryption and MAC operations are defined by the CipherSpec and constrained by `CipherSpec.bulk_cipher, .mac_function,` and `.key_exchange`. The encryption and MAC functions translate a `STLPCompressed` structure into a `STLPCiphertext`. The decryption functions reverse the process. Transmissions also use sequence numbers so that missing or replayed messages are detectable.

```
struct{
        select (ContentType)
              case change_cipher_spec:  opaque
                                    new_key<1..2^8 -1>;
              case application_datagram:  opaque
                                    new_key_seed<1..2^8 -1>;
```

```
} KeyInfoBlock;

struct {
        ContentType type;
        ProtocolVersion version;
        uint16 length;
             select (ContentType) {
                 case change_cipher_spec, alert, handshake,
                 application_data(23):;
                 case application_datagram:  KeyInfoBlock;
             } key_info;
             select (CipherSpec.cipher_type) {
                 case stream: GenericStreamCipher;
                 case block: GenericBlockCipher;
             } fragment;
             opaque MAC[CipherSpec.hash_size];
     } STLPCiphertext;
```

`type, version`
These type fields are identical to `STLPPlaintext.type` and `.version`.
`length`
The length (in bytes) of the following `STLPCiphertext.fragment`.
The length may not exceed $2^{14} + 2048$.
`key_info`
For records of type `application_datagram`, some key information must be delivered along with the record. This is contained in the `key_info` field. Derivation of an encryption and MAC key using this field is covered in section 6.
`fragment`
The encrypted form of `STLPCompressed.fragment`.
`MAC`
The message authentication code (MAC) included to guarantee message authenticity.

The MAC is generated as

```
MAC = Hash ( MAC_write_secret + Hash
(ConnectionParams.cipher_spec_counter + ConnectionParams.seq_num
+ STLPCiphertext));
```

where "+" denotes concatenation.

`cipher_spec_counter`
A counter that increments every time a new encryption key is delivered using a **change cipher spec** message. Thus, MAC inputs for different streams of pre-encrypted data are kept distinct. (See section 4.3.)
`seq_num`
The sequence number for this message. Messages of type `application_datagram` always have a `seq_num` value of zero.
`Hash`
The hashing algorithm determined by `CipherSpec.cipher_list.hash`.

Note that the MAC is computed after encryption.

4.2.3.1 Null or standard stream cipher

Stream ciphers (including NULL - See Appendix A) convert STLPCompressed.fragment structures to and from stream STLPCiphertext.fragment structures.

```
stream-ciphered struct {
       opaque content[STLPCompressed.length];
} GenericStreamCipher;
```

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 CipherSpec.bulk_encryption_algorithm is NULL, then 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). STLPCiphertext.length is STLPCompressed.length plus CipherSpec.hash_size (plus the length of key_info, for ContentType application_datagram only).

4.2.3.2 CBC block cipher

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

```
block-ciphered struct {
        opaque content[STLPCompressed.length];
        uint8 padding[GenericBlockCipher.padding_length];
        uint8 padding_length;
} GenericBlockCipher;
```

padding
Padding that is added to force the length of the plaintext to be a multiple of the block cipher's block length.

padding_length
The length of the padding must be less than the cipher's block length and may be zero. The padding length should be such that the total size of the GenericBlockCipher structure is a multiple of the cipher's block length.

The encrypted data length (STLPCiphertext.length) is one more than the sum of STLPCompressed.length, CipherSpec.hash_size, padding_length and the length of key_info (for ContentType application_datagram only).

Note: With CBC block chaining the initialization vector (IV) for the first record is provided by the handshake protocol. The IV for subsequent records is the last ciphertext block from the previous record.

4.3 The 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*) CipherSpec. This protocol is used in two circumstances: during a handshake, to signal the adoption of a newly negotiated CipherSpec; and to cause a temporary new key to be incorporated into a new CipherSpec (and then later to cause the old one to be restored) in order to transmit pre-encrypted data.

```
struct {
        enum { change_cipher_spec(1), (255) } type;
        CipherList new_cipher_list;
        KeyInfoBlock new_key_info;
} ChangeCipherSpec;
```

`new_cipher_list`

The correct new value for `CipherSpec.cipher_list`. This field must contain exactly one choice in each of its three sublists; these in turn must have been contained in the corresponding sublists in the `hello` message received in the current (or most recent) handshake. (See section 5.2.1.)

`new_key_info`

The new key to be used to encrypt and decrypt data sent in the same direction as the **change cipher spec** message containing it. This field is not used when the message is sent as part of a handshake.

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 just-negotiated CipherSpec and keys. Reception of this message causes the receiver to copy the *read pending* state into the *read current* state. Separate read and write states are maintained by both the `STLP` client and server. When the client or server receives a **change cipher spec** message, it copies the pending read state into the current read state. When the client or server writes a **change cipher spec** message, it copies the pending write state into the current write state. The client sends a **change cipher spec** message following the handshake `key_exchange` and `certificate_verify` messages (if any), and the server sends one after successfully processing the `key_exchange` message it received from the client. When resuming a previous session, the **change cipher spec** message is sent after both hello messages have been sent and received.

When a **change cipher spec** message containing a non-empty `new_key_info` field is sent and received after the handshake has already been completed, the sender's current write state and receiver's current read state are saved, then replaced with new ones containing the `new_cipher_list` as the `cipher_list` and the `new_key` value replacing the sender's write key (for both sender and receiver). If the `new_key_info` field is empty, and a previous sender's write state and receiver's read state have been saved, then they are restored when the **change cipher spec** message is sent and received. Only the initial states following the handshake are ever saved; if there are already states saved when a **change cipher spec** message is sent and received, then those states continue to be saved, and the current ones are discarded.

Note: Sequence numbers continue to be incremented for saved states; sequence numbers for new states begin at zero. In addition, the `ConnectionParams.cipher_spec_counter` field in the CipherSpec is incremented in both the new current state and the saved one, if any, every time a new **change cipher spec** message is received with a non-empty `new_key_info` field. If a **change cipher spec** message is received with an empty `new_key_info` field, then this counter's value is reset to zero.

4.4 Alert protocol

One of the content types supported by the STLP 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),
        decompression_failure(30),
        handshake_failure(40), no_certificate(41),
bad_certificate(42),
            unsupported_certificate(43),
certificate_revoked(44),
            certificate_expired(45), certificate_unknown(46),
            illegal_parameter (47),
        (255)
} AlertDescription;

enum { cipher_mismatch(01), hash_mismatch(02),
exchange_mismatch(04), signature_mismatch(08),
certificate_mismatch(16), certifier_mismatch(32),
combination_mismatch(64), authentication_mismatch(128),(255)
} AlertDetails;



struct {
        AlertLevel level;
        AlertDescription description;
        AlertDetails details<0..2>;
} Alert;
```

4.4.1 Closure alerts

The client and the server must share knowledge that the connection is ending in order to avoid a truncation attack, in which an adversary prematurely closes a connection, by making connection closure explicit and cryptographically authenticated. 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.

4.4.2 Error alerts

Error handling in the STLP Handshake protocol is very simple. When an error is detected, the detecting party sends an `alert` 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.

`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 attributes given the options available. This is a fatal error. This **alert** message and the `no_certificate` message are the only two in which the `Alert.details` field is non-empty; the logical (inclusive) OR of the appropriate `AlertDetails` values is sent to indicate which types of security attribute produced an unavoidable mismatch.

`no_certificate`
A `no_certificate` **alert** message may be sent in response to a `certification_request` if no appropriate certificate is available. This **alert** message and the `handshake_failure` message are the only two in which the `Alert.details` field is non-empty; the (inclusive) logical OR of the appropriate `AlertDetails` values is sent to indicate which types of security attribute produced an unavoidable mismatch. The value `combination_mismatch` indicates that no particular parameter produced a mismatch, but that no acceptable combination could be found. Such a mismatch occurs when no certificate is available which matches the certificate requester’s choices of certificate type, signature type and certifier simultaneously.

`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.

The `Alert.details` field is empty except in the case of a `handshake_failure` or `no_certificate` error.

5. Handshake Protocol

5.1 Handshake protocol overview

The cryptographic parameters of the session state are produced by the **STLP Handshake Protocol**, which operates on top of the **STLP Record Layer**. When a STLP 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. These processes are performed in the handshake protocol, which can be summarized as follows: Either the client or the server sends a **hello** message of type **initiating,** to which the other must respond with a **hello** message of type **responding**, or else a fatal error will occur and the connection will fail. The two **hello** messages are used to establish security enhancement capabilities between client and server, establishing such attributes as protocol version, session ID, supported cipher and certificate types, and compression method. Additionally, two random challenge values are generated and exchanged. Finally, the two messages establish which of the client or server (or both) are to be authenticated, and by what method—key exchange, digital signature or shared password.

Accompanying the hello messages, or following later in the protocol, will be **public value, client key exchange,** and **response** messages. (Messages may be grouped together and sent as one; a **finished** message is always sent at the end of such a group to indicate that no more handshake messages will be sent before at least one handshake message is first received.) For example, the client may send **public value** and **client key exchange** messages along with its **hello** message, if it has the server's key exchange public value from some other source. These messages effect a key exchange between client and server, using the previously-known server's public value (sent in the **public value** message to indicate which one is being used). If the client used the wrong public value, or if the server is sending the first **hello** message, then the server sends its key exchange public value in a **public value** message; the client then responds with a **client key exchange** message completing the key exchange.

The server may authenticate in one of three ways. To authenticate using key exchange, the server sends a **response** message following the **client key exchange** message, containing a keyed hash value (using a collision-intractable hash function, and a key derived from the one just exchanged) of all handshake messages sent up to the response message. This authentication method requires that the public value used in the key exchange be certified. To authenticate using digital signature, the server sends **public value** and **response** messages (in that order) following both **hello** messages and a **public value** message containing a key exchange public value. The the second **public value** message contains a certified signature key, and the **response** message contains a digital signature (using the certfied signature key) of all the handshake messages sent up to the **response** message. To authenticate using a shared password, the server sends a **response** message following the **client key exchange** message and client **response** message, containing a keyed hash value (using a collision-intractable hash function, and a key derived from the one just exchanged) of the server's identity and password, together with all the handshake messages sent up to the **response** message. (Note that the client must authenticate first before the server may authenticate by shared password.)

The client may authenticate by either digital signature or password. Digital signature-based client authentication is identical to the server case, except that the **public value** and **response** messages are sent along with the **client key exchange** message. Client authentication by shared password is also identical to the server case except for the timing of the **response** message, which is sent following the **client key exchange** message, provided the server has already sent a **response** message or a **public value** message from the server contained a certified key exchange public value.

A **change cipher spec** message is also sent by the client accompanying the **client key exchange** message, immediately before the **finished** message; at this point, the client copies the *pending* CipherSpec into the *current* CipherSpec. 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* CipherSpec, and send its **Finished** message under the new CipherSpec. At this point, the handshake is complete and the client and server may begin to exchange application layer data.

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

For example, if the client initiates the session, and the server authenticates by key exchange, the protocol would have the following flow:

Client	**Server**
hello	
finished	
	hello
	public value
	finished

Client	Server
client key exchange	
change cipher spec	
finished	
	response
	change cipher spec
	finished

If the client already has the server's correct key exchange public value, then the protocol flow is as follows:

Client	**Server**
hello	
public value	
client key exchange	
finished	
	hello
	response
	change cipher spec
	finished
change cipher spec	
finished	

If the server is authenticating by digital signature, and client authentication is required (by either digital signature or shared password), the protocol flow is as follows:

Client	**Server**
hello	
finished	
	hello
	public value
	response
	finished
client key exchange	
public value	
response	
change cipher spec	
finished	
	change cipher spec
	finished

If the server initiates the handshake, and authenticates by key exchange, then the following protocol flow occurs:

Client	**Server**
	hello

Client	Server
	public value
	finished
hello	
client key exchange	
change cipher spec	
finished	
	response
	change cipher spec
	finished

Finally, if client authentication (by digital signature or password) is required, and the (handshake-initiating) server's key exchange public value is certified, then the following protocol flow occurs:

Client	Server
	hello
	public value
	finished
hello	
client key exchange	
public value	
response	
change cipher spec	
finished	
	response
	change cipher spec
	finished

These examples are not exhaustive; other combinations are possible. Data transmission may begin as soon as the **change cipher spec** message has been sent.

When the client or server attempts to resume a previous session or duplicate an existing session (instead of negotiating new security parameters) the message flow is as follows: The initiator sends a **hello** message which includes the Session ID of the session to be resumed. The responder then checks its session cache for a match. If a match is found, and the responder is willing to re-establish the connection under the specified session state, it will send a **hello** message with the same Session ID value, and an authenticating **response** message. At this point, both send **change cipher spec** and **finished** messages (responder first). Once these have been sent, the client and server may begin to exchange application layer data. If a Session ID match is not found, the responder generates a new session ID and the STLP client and server perform a full handshake. The successful reconnection has the following message flow if the client initiates it:

Client	Server
hello	
finished	
	hello
	response
	change cipher spec

finished

response
change cipher spec
finished

If the server initiates the handshake, then the same messages are sent, with the senders reversed. The contents and significance of each message will be presented in detail in the following sections.

5.2 Handshake protocol

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

```
enum {
        hello(1), public_value(11), client_key_exchange(16),
        finished(20), response(30), (255)
} HandshakeType;

struct {
        HandshakeType msg_type;         /* type of handshake
message */
        uint24 length;  /* # bytes in handshake msg body */
        select (HandshakeType) {
            case hello:  Hello;
            case public_value: PublicValue;
            case client_key_exchange: ClientKeyExchange;
            case finished: Finished;
            case response:  Response;
        } body;
} Handshake;
```

5.2.1 Hello message

The hello phase messages are used to exchange security enhancement capabilities between the client and server. When a new session begins, the CipherSpec encryption, hash, key exchange and compression algorithms are initialized to `NULL`. The current CipherSpec is used for renegotiation messages.

```
enum {InitiatingClient(01), InitiatingServer(02),
      RespondingClient(03), RespondingServer(04), (255)}
} HelloType;
```

When a client first connects to a server, or vice versa, the initiator of the connection is required to send a **hello** message as its first message. The role played by the initiator (client or server) is indicated by the `hello_type` field. A **hello** message may also be sent at any time during the protocol to initiate a new handshake negotiation (unless the handshake associated with a previously sent **hello** message has not yet completed). The **hello** message includes a random structure, which is used later in the protocol.

```
struct {
   opaque random_bytes[32];
} Random;
```

`random_bytes`
32 bytes generated by a secure random number generator.

The initiating **hello** message includes a variable length session identifier, which may identify a session between the same client and server whose security parameters the initiator 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 simultaneous independent secure connections without repeating the full handshake protocol. If the initiator does not wish to reuse a previously established session, then the session identifier field is left empty.

The responder looks in its session cache for a session identifier matching the one supplied by the initiator. If a match is found and the responder 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 **response** and **change cipher spec** messages. Otherwise this field will contain a different random 32-byte value identifying the new session. (It is important that it be random, to avoid collisions with session identifiers already in use by the responder.)

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

Warning: Servers must not place confidential information in session identifiers or let the contents of fake session identifiers cause any breach of security.

The **authentication options** field consists of a set of flags indicating the set of authentications required and offered by both parties. The initiating sender of a **hello** message computes the logical (inclusive) OR of all the types of responder authentication acceptable to the initiator, and all the types of initiator authentication that the initiator is willing to provide to the responder. For example, an initiating client that requires server authentication by key exchange, and is prepared to authenticate by signature or by shared password, would compute the value of the **authentication options** field as the logical (inclusive) OR of the `ServerKeyExchange`, `ClientSignature` and `ClientPassword AuthType` values. If none of the responder authentication types are chosen by the initiator, then no authentication of the responder is required.

The responder's **hello** message makes the final authentication choices, by selecting one of the initiator authentication options offered (if desired), and one of the responder authentication options requested (if any), by the initiator. If the choices offered by the initiator are unacceptable to the responder, then the connection is closed with a `handshake_failure` alert, and a value of `authentication_mismatch` in the `details` field.

```
enum {ServerKeyExchange(01), ServerSignature(02),
ServerPassword(04), ClientSignature(08), ClientPassword(16),
ServerReconnect(32), ClientReconnect(64),(255)
}AuthType;
```

The **hello** message also includes a list of compression algorithms supported by the initiator, ordered according to the initiator's preference. The responder selects its choice from this list and returns it in the subsequent **hello** message. (There is always at least one acceptable choice, since compression method `NULL` must always be supported.)

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

The `cipher_list and cert_list` list sets passed from the initiator to the responder in the initiator's **hello** message contain lists of cryptographic algorithms and certificate types supported by the initiator in order of the initiator's preference (first choice first). The `cipher_list` consists of a `symmetric` (cipher), `hash` (algorithm) and `key_exchange` (algorithm) list. The responder selects a symmetric cipher, hash algorithm and key exchange algorithm from these lists, or, if no acceptable choices are presented, returns a `handshake_failure` alert with the appropriate `details` value and closes the connection. Similarly, the initiator's `cert_list` consists of a `signature` (algorithm), `certificate` (type) and `certifier` (name) list. The responder responds with a subset of each of these lists, or, if no acceptable choices are presented, returns a `handshake_failure` alert with the appropriate `details` value and closes the connection. Both client and server can use values in the other's `cert_list` lists to determine which certificate to deliver to the other if necessary for authentication purposes. In the case of the `cipher_list`, each responder's sublist must contain exactly one entry; this set of single entries is copied into the `CipherSpec.cipher_list` field.

The `symmetric, hash, key_exchange, signature` and `certificate` sublists contain enumerated types; the `certifier` list contains zero-delimited strings (such as X.509 distinguished names). An empty `certifier` list simply indicates that no particular certifier is specified by the sender; instead, certificates' certifiers will be evaluated upon receipt.

```
enum { NULL(00), DES(01), DES_112(02), DES_168(03),
DES_40(04), IDEA(05), RC2_128(06), RC2_40(07), RC4_128(08),
RC4_40(09), Fortezza(10), (255)
} SymmetricCipherAlgorithm;

enum { NULL(00), MD5(01), MD5_TRUNC_64(02), SHA(03),
SHA_TRUNC_80(04),(255)
} HashAlgorithm;

enum { rsa(01), diffie_hellman(02), fortezza_dms(03), (255) }
KeyExchangeAlgorithm;

enum { RSA_MD5(01), RSA_SHA(02), DSS_SHA(03), (255)
} SignatureAlgorithm;
```

```
enum { NULL(00), Private(01), X509(02), PKCS_7(03), (255)
 } CertificateType;


opaque CertifierName<0..2^16-1>;

struct {
        SymmetricCipherAlgorithm symmetric<1..2^8 -1>;
        HashAlgorithm hash<1..2^8 -1>;
        KeyExchangeAlgorithm key_exchange<1..2^8 -1>;
} CipherList;

struct {
        SignatureAlgorithm signature<0..2^8-1>;
        CertificateType cert_type<1..2^8 -1>;
        CertifierName certifier<0..2^16-1>;
} CertList;

struct {
        HelloType sender;
        ProtocolVersion version;
        Random random;
        SessionID session_id;
        uint8 auth_choices;
        CipherList cipher_list;
        CertList cert_list;
        CompressionMethod compression_methods<2..2^8 -1>;
        Opaque trailing_data<0..2^16-1>;
} Hello;
```

`sender`
The sender of the **hello** message.

`version`
The version of the STLP protocol by which the sender wishes to communicate during this session. This should be the most recent (highest valued) version supported by the sender. For this version of the specification, the version will be 1. The version used for the actual protocol is the minimum of the two versions sent in the two **hello** messages.

`random`
A sender-generated random structure.

`session_id`
The identifier of a session the sender wishes to use for this connection. In the case of an initiator's **hello** message, this field should be *empty* if no session identifier is available or the sender wishes to establish a new session. In the case of a responder's **hello** message, this field should echo the session identifier in the initiator's **hello** message if it is recognized and acceptable to the responder; otherwise, this field should contain a random 32-byte value.

`auth_choices`
In the initiator's message, the initiator's choices for the responder's authentication type and the responder's options for the initiator's authentication type. In the responder's message, the

responder’s choices for authentication of both initiator and responder, taken from the choices provided by the initiator.

`cipher_list`
In the case of an initiator’s **hello** message, this field contains a set of lists of the cryptographic options supported by the sender (symmetric cipher, hash algorithm and key exchange algorithm), sorted with the initiator's first preference first. In the case of a responder’s **hello** message, this field contains the responder’s choice from each list in the initiator’s message.

`cert_list`
A set of three lists indicating the types of signature algorithm and certificate, and the certifiers (by name) supported by the sender. These are used by the receiver to determine which type of certificate to send, if necessary, for authentication purposes.

`compression_methods`
In the case of an initiator’s message, this is a list of the compression methods supported by the sender, sorted by client preference (and always including the compression method `NULL`). In the case of a responder’s message, this is a choice from the initiator’s list.

After sending the **hello** message, the initiator sends a **finished** message, then waits for the responder’s **hello** message. Any other handshake message returned by the responder is treated as a fatal error.

Implementation note: Application data may not be sent before a **change cipher spec** message has been sent. Transmitted application data is known to be insecure until a valid **change cipher spec** message has been received. This absolute restriction is relaxed if there is a current, non-null encryption on this connection.

5.2.2 Finished message

Handshake messages are grouped together by the presence of a **finished** message at the end of every group. The sender of a sequence of one or more consecutive handshake messages appends a **finished** message to the end of the sequence; the receiver then can recognize the end of a such a sequence, and determine when to respond with its own sequence of handshake messages.

The **finished** handshake message has no fields or values other than its message type.

```
struct {
} Finished;
```

5.2.3 Public Value message

The server normally sends its certificate (or uncertified public value) in a **public value** message immediately following both **hello** messages. If the server is authenticating by key exchange, then the **public value** message will contain a certified key exchange public key (`PublicValueType key_exchange_certificate`) of a type indicated in the client’s **hello** message as acceptable. If the server is authenticating by signature, then two **public value** messages are sent, one containing the (uncertified) key exchange public key to be used for key exchange and the other the certified signature public key key (`PublicValueType signature_certificate`) used for authentication. If the server is authenticating by shared password, then the **public value** message contains an identifier recognizeable to the client, passed as a public value of type `identity`. The interpretation of this field is left to the implementation.

If the client is authenticating by digital signature, then it sends a **public value** message containing a certified signature public key (`PublicValueType signature_certificate`). This message may be sent at any time following both **hello** messages. Similarly, if the client is authenticating by password, then it sends a **public value** message containing an identifier recognizeable to the client, passed as a public value of type `identity`. The interpretation of this field is left to the implementation.

```
enum {rsa, RSA_PKCS, diffie_hellman, DH_PKCS, fortezza,
signature_certificate, key_exchange_certificate, identity
} PublicValueType;

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

`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>;
} DHParams;
```

`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 {
            opaque r_s [128];
} FortezzaParams;
```

`r_s`
Server random number for Fortezza KEA (Key Exchange Algorithm).

```
struct {
     PublicValueType public_value_label;
```

```
      CertificateType cert_type;
      select public_value_label
          case signature_certificate, key_exchange_certificate:
                    opaque certificate_list<1..2^24 -1>;
          case RSA_PKCS:  opaque rsa_pkcs_public_value<1..2^16-1>;
          case DH_PKCS:  opaque dh_pkcs_public_value<1..2^16-1>;
          case rsa:  RSAParams rsa_public_value;
          case dh:   DHParams dh_public_value;
          case fortezza:  FortezzaParams fortezza_public_value;\
          case identity:  opaque id;
} PublicValue;
```

`public_value_label`
The type of public value contained in the public value message.

`cert_type`
The type of certificate in the public value message; if the message does not contain a certificate, then this field has value `NULL`.

`certificate_list`
If the certificate type is X509, this is a sequence (chain) of X.509.v3 certificates, ordered with the sender's certificate first and the root certificate authority last.

`rsa_PKCS_public_value`
An RSA public value, in PKCS format.

`dh_pkcs_public_value`
A Diffie-Hellman public value, in PKCS format.

`identity`
An identity label associated with a particular shared password; the interpretation of the identity label is left to the implementation.

5.2.4 Client Key Exchange message

The format of this message depends on which public key algorithm(s) has (have) been selected.

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

The information to select the appropriate record structure is in the *pending* session state.

5.2.4.1 RSA encrypted premaster secret format

If RSA is being used for key agreement and authentication, the client generates a 48-byte pre-master secret, encrypts it under the public key from the server's certificate or temporary RSA key from the server's **public value** message, and sends the result in an encrypted premaster secret format.

```
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.

`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 6.

5.2.4.2 Fortezza key exchange message

Under Fortezza DMS, the client derives a Token Encryption Key (TEK) using Fortezza's Key Exchange Algorithm (KEA). The client's KEA calculation uses the public key in the server's certificate along with private parameters in the client's token. The client sends public parameters needed for the server to generate the TEK, using its own private parameters. The client generates session keys, wraps them using the TEK, and sends the results to the server. The client generates IV's for the session keys and TEK and sends them also. The client generates a random 48-byte premaster secret, encrypts it using the TEK, and sends the result:

```
struct {
        opaque y_c<0..128>;
        opaque r_c[128];
        opaque wrapped_client_write_key[12];
        opaque wrapped_server_write_key[12];
        opaque client_write_iv[24];
        opaque server_write_iv[24];
        opaque master_secret_iv[24];
        block-ciphered opaque encrypted_pre_master_secret[48];
} FortezzaKeys;
```

`y_c`

The client's Yc value (public value) for the KEA calculation. If the client has sent a **public value** message containing a certificate, and its KEA public key is suitable, this value must be *empty* since the certificate already contains this value. If the client sent no certificate, or a certificate without a suitable public key, y_c contains the public value. For this value to be used, it must be between 64 and 128 bytes.

`r_c`
The client's Rc value for the KEA calculation.

`wrapped_client_write_key`
This is the client's write key, wrapped by the TEK.

`wrapped_server_write_key`
This is the server's write key, wrapped by the TEK.

client_write_iv
This is the IV for the client write key.

`server_write_iv`
<DD>This is the IV for the server write key.

`master_secret_iv`
This is the IV for the TEK used to encrypt the pre-master
secret.

`pre_master_secret`
This is a random value, generated by the client and used to generate the **master secret**. In the above structure, it is encrypted using the TEK.

5.2.4.3 Client Diffie-Hellman public value

This structure conveys the client's Diffie-Hellman public value (Yc) if it was not already included in a certificate sent by the client. The encoding used for Yc is determined by the enumerated `PublicValueEncoding.`

```
enum { implicit, explicit } PublicValueEncoding;
```

`implicit`
If the client certificate already contains the public value, then it is implicit and Yc does not need to be sent again.

`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).

5.2.5 Response message

The **response** message is sent by either client or server to authenticate to the other. It consists of a single field whose contents are computed based on previous handshake messages and some secret information only available to the party being authenticated.

Note: Since the **response** message is always sent after both **hello** messages, it also serves as a verification that the negotiation of cipher and certificate properties has not been tampered with.

```
struct {
       AuthType sender_auth;
       select (sender_auth) {
       case ServerKeyExchange, ServerPassword, ClientPassword,
ServerReconnect, ClientReconnect:
            opaque response_hash[CipherSpec.hash_size];
       case ServerSignature, ClientSignature:  digitally signed
            opaque signature_response[CipherSpec.hash_size];
         }
} Response;
```

`sender_auth`
The type of authentication being performed with this **response** message.

`Response_hash`
In the case of key exchange-based or password-based authentication, the correct response is simply a keyed hash of all the handshake messages passed up until the **response** message (including the other party's **response** message, if already sent). The keyed hash is computed as a MAC of all of these messages, with their concatenation replacing the `STLPCiphertext` field, and the two counter inputs' values set to zero. (The hash function used for MAC computation is determined by the pending cipher state.) The value of `Response.auth_type` is appended to the messages before input into the hash function; for password-based authentication, the shared password is appended as well, after the `auth_type` value. More explicitly, the response is computed as:

```
opaque password<0..2^16 - 1>;
uint64 zeroes;
```

```
zeroes = 0;

Response_hash = Hash ( MAC_write_secret + Hash
(zeroes + handshake messages + Response.auth_type + password))
```

(For non-password-based authentication, the `password` field is omitted.)

`signature_response`
In the case of signature-based authentication, the correct response is computed as the digital signature of the "inner hash" of the MAC computation described above. That is, the digital signature algorithm (including, typically, a hash function invocation) is performed on

```
Hash (zeroes + handshake messages + Response.auth_type)
```

The private signature key used is the one associated with the public value passed by the sender in a previous **public value** message.

6. Cryptographic computations

The key exchange, authentication, encryption, and MAC algorithms are determined by the `cipher_list` negotiated during the handshake.

6.1 Asymmetric cryptographic computations

The asymmetric algorithms are used in the handshake protocol to authenticate parties and to generate shared keys and secrets.

For Diffie-Hellman, RSA, and Fortezza, 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 =
        Hash('A' + pre_master_secret + ClientHello.random +
ServerHello.random) + Hash('BB' + pre_master_secret +
ClientHello.random + ServerHello.random) + Hash('CCC' +
pre_master_secret + ClientHello.random + ServerHello.random)[…];
```

until 48 bytes of output has been generated (extra output at the end is ignored). The hash algorithm `Hash` is determined by the value of `CipherSpec.cipher_list.hash` in the pendng CipherSpec.

6.1.1 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.

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

6.1.3 Fortezza

A random 48-byte `pre_master_secret` is sent encrypted under the TEK and its IV. The server decrypts the `pre_master_secret` and converts it into a `master_secret`, as specified above. Bulk cipher keys and IVs for encryption are generated by the client's token and exchanged in the **client key exchange** message; the `master_secret` is only used for MAC computations.

6.2 Symmetric cryptographic calculations and the CipherSpec

The technique used to encrypt and verify the integrity of STLP records is specified by the currently active CipherSpec. A typical example would be to encrypt data using DES and generate authentication codes using MD5. The encryption, hash and key exchange algorithms are set to `NULL` at the beginning of the STLP Handshake Protocol, indicating that no message authentication or encryption is performed. The handshake protocol is used to negotiate a more secure CipherSpec and to generate cryptographic keys.

6.2.1 The master secret

Before secure encryption or integrity verification can be performed on records, the client and server need to generate shared secret information known only to themselves. This value is a 48-byte quantity called the *master secret*. The master secret is used to generate keys and secrets for encryption and MAC computations. Some algorithms, such as Fortezza, may have their own procedure for generating encryption keys (the master secret is used only for MAC computations in Fortezza).

6.2.2 Converting the master secret into keys and MAC secrets

The master secret is used to generate the secrets, keys, and non-export IVs required by the current CipherSpec (see Appendix A).

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, such as Fortezza keys communicated in the KeyExchange message, are *empty*. The following inputs are available to the key definition process:

```
CipherSpec.master_secret
ConnectionParams.ClientHelloRandom
ConnectionParams.ServerHelloRandom;
```

The `ClientHelloRandom` and `ServerHelloRandom` values are obtained from the **Hello** messages for the current connection, and are part of the current connection state.

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. In the case of records of type `application_datagram`, the `master_secret` is processed one more time before being used for key derivation, as follows:

```
new_master_secret = Hash('A' + key_info.new_key_seed +
master_secret ) + Hash('BB' + key_info.new_key_seed +
master_secret ) + = Hash('CCC' + key_info.new_key_seed +
master_secret ) + […];
```

until 48 bytes of output has been generated (extra output at the end is ignored). The hash algorithm `Hash` is determined by the value of `CipherSpec.cipher_list.hash` in the current CipherSpec. The resulting value of `new_master_secret` is used in place of `master_secret` in the key derivation for the datagram record.

To generate the key material, compute

```
key_block = Hash('A' + master_secret + ServerHelloRandom +
ClientHelloRandom) + Hash('BB' + master_secret +
ServerHelloRandom + ClientHelloRandom) + Hash('CCC' +
master_secret + ServerHelloRandom + ClientHelloRandom) + […];
```

until enough output has been generated (extra output at the end is ignored). The hash algorithm Hash is determined by the value of CipherSpec.cipher_list.hash in the pendng CipherSpec. The key_block is then partitioned as follows:

```
client_write_MAC_secret[CipherSpec.hash_size]
server_write_MAC_secret[CipherSpec.hash_size]
client_write_key[CipherSpec.key_material]
server_write_key[CipherSPec.key_material]
client_write_IV[CipherSpec.IV_size] /* non-export ciphers */
server_write_IV[CipherSpec.IV_size] /* non-export ciphers */
```

Any extra `key_block` material is discarded. 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 = Hash('A' + client_write_key +
ClientHelloRandom + ServerHelloRandom) + Hash('BB' +
client_write_key + ClientHelloRandom + ServerHelloRandom) +
Hash('CCC' + client_write_key + ClientHelloRandom +
ServerHelloRandom) + […];

final_server_write_key = Hash('A' + server_write_key +
ServerHelloRandom + ClientHelloRandom) + Hash('BB' +
server_write_key + ServerHelloRandom + ClientHelloRandom) +
Hash('CCC' + server_write_key + ServerHelloRandom +
ClientHelloRandom) + […];
```

until enough output has been generated (extra output at the end is ignored). The hash algorithm `Hash` is determined by the value of CipherSpec.cipher_list.hash in the pendng CipherSpec.

Exportable encryption algorithms derive their IVs from the random messages:

```
client_write_IV = Hash('A', ClientHelloRandom +
ServerHelloRandom) + Hash('BB', ClientHelloRandom +
ServerHelloRandom) + Hash('CCC', ClientHelloRandom +
ServerHelloRandom) + […];

server_write_IV = Hash('A' + ServerHelloRandom +
ClientHelloRandom) + Hash('BB' + ServerHelloRandom +
ClientHelloRandom) + Hash('CCC' + ServerHelloRandom +
ClientHelloRandom) + […];
```

until enough output has been generated (extra output at the end is ignored). The hash algorithm `Hash` is determined by the value of CipherSpec.cipher_list.hash in the pendng CipherSpec.

Appendix A. The CipherSpec and ConnectionParams

The CipherSpec is part of the STLP session state. The CipherSpec has the following structure:

```
enum { stream, block } CipherType;
enum { true, false } IsExportable;


struct {
 CipherList cipher_list;
        CipherType cipher_type;
        IsExportable is_exportable;
        uint8 hash_size;
        opaque master_secret[48];
        uint8 key_material;
        uint8 IV_size;
} CipherSpec;
```

The `ConnectionParams` structure is as follows:

```
struct {
        opaque ClientHelloRandom[32];
        opaque ServerHelloRandom[32];
        uint32 seq_num;
        uint32 cipher_spec_counter;
} ConnectionParams;
```