Internet Engineering Task Force (IETF)
Request for Comments: 6101
Category: Historic
ISSN: 2070-1721

A. Freier
P. Karlton
Netscape Communications
P. Kocher
Independent Consultant

August 2011

The Secure Sockets Layer (SSL) Protocol Version 3.0

Abstract

This document is published as a historical record of the SSL 3.0 protocol. The original Abstract follows.

This document specifies version 3.0 of the Secure Sockets Layer (SSL 3.0) protocol, a security protocol that 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.

Foreword

Although the SSL 3.0 protocol is a widely implemented protocol, a pioneer in secure communications protocols, and the basis for Transport Layer Security (TLS), it was never formally published by the IETF, except in several expired Internet-Drafts. This allowed no easy referencing to the protocol. We believe a stable reference to the original document should exist and for that reason, this document describes what is known as the last published version of the SSL 3.0 protocol, that is, the November 18, 1996, version of the protocol.

There were no changes to the original document other than trivial editorial changes and the addition of a "Security Considerations" section. However, portions of the original document that no longer apply were not included. Such as the "Patent Statement" section, the "Reserved Ports Assignment" section, and the cipher-suite registrator note in the "The CipherSuite" section. The "US export rules" discussed in the document do not apply today but are kept intact to provide context for decisions taken in protocol design. The "Goals of This Document" section indicates the goals for adopters of SSL 3.0, not goals of the IETF.

The authors and editors were retained as in the original document. The editor of this document is Nikos Mavrogiannopoulos (nikos.mavrogiannopoulos@esat.kuleuven.be). The editor would like to thank Dan Harkins, Linda Dunbar, Sean Turner, and Geoffrey Keating for reviewing this document and providing helpful comments.

Status of This Memo

This document is not an Internet Standards Track specification; it is published for the historical record.

This document defines a Historic Document for the Internet community. This document is a product of the Internet Engineering Task Force (IETF). It represents the consensus of the IETF community. It has received public review and has been approved for publication by the Internet Engineering Steering Group (IESG). Not all documents approved by the IESG are a candidate for any level of Internet Standard; see Section 2 of RFC 5741.

Information about the current status of this document, any errata, and how to provide feedback on it may be obtained at http://www.rfc-editor.org/info/rfc6101.

Copyright Notice

Table of Contents

1. Introduction

The primary goal of the SSL protocol is to provide privacy and reliability between two communicating applications. The protocol is composed of two layers. At the lowest level, layered on top of some reliable transport protocol (e.g., TCP [RFC0793]), is the SSL record protocol. The SSL record protocol is used for encapsulation of various higher level protocols. One such encapsulated protocol, the SSL 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. One advantage of SSL is that it is application protocol independent. A higher level protocol can layer on top of the SSL protocol transparently. The SSL protocol provides connection security that has three basic properties:

- The connection is private. Encryption is used after an initial handshake to define a secret key. Symmetric cryptography is used for data encryption (e.g., DES [DES], 3DES [3DES], RC4 [SCH]).

- The peer's identity can be authenticated using asymmetric, or public key, cryptography (e.g., RSA [RSA], DSS [DSS]).

- The connection is reliable. Message transport includes a message integrity check using a keyed Message Authentication Code (MAC) [RFC2104]. Secure hash functions (e.g., SHA, MD5) are used for MAC computations.

2. Goals

The goals of SSL protocol version 3.0, in order of their priority, are:

1. Cryptographic security

   SSL should be used to establish a secure connection between two parties.

2. Interoperability

   Independent programmers should be able to develop applications utilizing SSL 3.0 that will then be able to successfully exchange cryptographic parameters without knowledge of one another's code.

Note: It is not the case that all instances of SSL (even in the same application domain) will be able to successfully connect. For instance, if the server supports a particular hardware token, and the client does not have access to such a token, then the connection will not succeed.

3. Extensibility

SSL 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 SSL 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

The SSL protocol version 3.0 specification is intended primarily for readers who will be implementing the protocol and those doing cryptographic analysis of it. The spec 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 or 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 External Data Representation (XDR) [RFC1832] in both its syntax and intent, it

would be risky to draw too many parallels. The purpose of this presentation language is to document SSL 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 byte stream, 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.

```
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];
```

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;
```

Optionally, one may 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, an

```
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 5.1).

In digital signing, one-way hash functions are used as input for a signing algorithm. In RSA signing, a 36-byte structure of two hashes (one SHA and one MD5) is signed (encrypted with the private key). In DSS, the 20 bytes of the SHA hash are run directly through the Digital Signature Algorithm with no additional hashing.

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, and vice versa. 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.

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. SSL Protocol

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

5.1. Session and Connection States

An SSL session is stateful. It is the responsibility of the SSL 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 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 5.3), and they then communicate using the newly agreed-upon cipher spec.

An SSL 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: X509.v3 [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.7 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.

The connection state includes the following elements:

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

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 Cipher Block Chaining (CBC) mode is used, an initialization vector (IV) is maintained for each key. This field is first initialized by the SSL handshake protocol. Thereafter, the final ciphertext block from each record is preserved for use with the following record.

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 uint64 and may not exceed 2^64-1.

5.2. Record Layer

The SSL record layer receives uninterpreted data from higher layers in non-empty blocks of arbitrary size.

5.2.1. Fragmentation

The record layer fragments information blocks into SSLPlaintext 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 SSLPlaintext record).

```
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[SSLPlaintext.length];
} SSLPlaintext;
```

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

version: The version of protocol being employed. This document describes SSL version 3.0 (see Appendix A.1).

length: The length (in bytes) of the following SSLPlaintext.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 SSL record layer content types may be interleaved. Application data is generally of lower precedence for transmission than other content types.

5.2.2. 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 an SSLPlaintext structure into an SSLCompressed structure. Compression functions erase their state information whenever the CipherSpec is replaced.

Note: The CipherSpec is part of the session state described in Section 5.1. 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.7.

Compression must be lossless and may not increase the content length by more than 1024 bytes. If the decompression function encounters an SSLCompressed.fragment that would decompress to a length in excess of 2^14 bytes, it should issue a fatal decompression_failure alert (Section 5.4.2).

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

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

fragment: The compressed form of SSLPlaintext.fragment.

Note: A CompressionMethod.null operation is an identity operation; no fields are altered (see Appendix A.4.1.)

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

5.2.3. 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, initially it is SSL_NULL_WITH_NULL_NULL, which does not provide any security.

Once the handshake is complete, the two parties have shared secrets that 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.cipher_type. The encryption and MAC functions translate an SSLCompressed structure into an SSLCiphertext. The decryption functions reverse the process. Transmissions also include a sequence number so that missing, altered, or extra messages are detectable.

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

type: The type field is identical to SSLCompressed.type.

version: The version field is identical to SSLCompressed.version.

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

fragment: The encrypted form of SSLCompressed.fragment, including the MAC.

5.2.3.1. Null or Standard Stream Cipher

Stream ciphers (including BulkCipherAlgorithm.null; see Appendix A.7) convert SSLCompressed.fragment structures to and from stream SSLCiphertext.fragment structures.

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

The MAC is generated as:

```
hash(MAC_write_secret + pad_2 +
     hash(MAC_write_secret + pad_1 + seq_num +
          SSLCompressed.type + SSLCompressed.length +
          SSLCompressed.fragment));
```

where "+" denotes concatenation.

pad_1: The character 0x36 repeated 48 times for MD5 or 40 times for SHA.

pad_2: The character 0x5c repeated 48 times for MD5 or 40 times for SHA.

seq_num: The sequence number for this message.

hash: Hashing algorithm derived from the cipher suite.

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 SSL_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). SSLCiphertext.length is SSLCompressed.length plus CipherSpec.hash_size.

5.2.3.2. CBC Block Cipher

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

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

The MAC is generated as described in Section 5.2.3.1.

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 (SSLCiphertext.length) is one more than the sum of SSLCompressed.length, CipherSpec.hash_size, and padding_length.

Note: With CBC, 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.

5.3. 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. 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 just-negotiated CipherSpec and keys. Reception of this message causes the receiver to copy the read pending state into the read current state. The client sends a change cipher spec message following 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. An unexpected change cipher spec message should generate an unexpected_message alert (Section 5.4.2). When resuming a previous session, the change cipher spec message is sent after the hello messages.

5.4. Alert Protocol

One of the content types supported by the SSL 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;

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

5.4.1. 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.

NB: It is assumed that closing a connection reliably delivers pending data before destroying the transport.

5.4.2. Error Alerts

Error handling in the SSL 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 a 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 parameters given the options available. This is a fatal error.

no_certificate: A no_certificate alert message may be sent in response to a certification request if no appropriate certificate is available.

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.

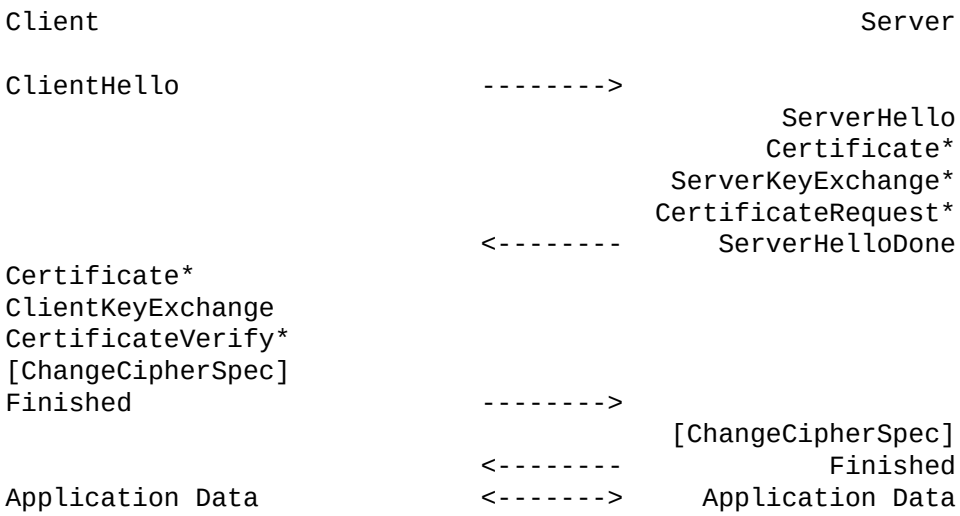
illegal_parameter: A field in the handshake was out of range or inconsistent with other fields. This is always fatal.

5.5. Handshake Protocol Overview

The cryptographic parameters of the session state are produced by the SSL handshake protocol, which operates on top of the SSL record layer. When an SSL 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: 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.

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 either the certificate message or a no_certificate alert. 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 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. (See flow chart below.)



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

Note: To help avoid pipeline stalls, ChangeCipherSpec is an independent SSL protocol content type, and is not actually an SSL 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 SSL client and server perform a full handshake.

```
      Client                                               Server

      ClientHello                  -------->
                                                      ServerHello
                                                [change cipher spec]
                                   <--------             Finished
      change cipher spec
      Finished                     -------->
      Application Data             <------->     Application Data
```

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

5.6. Handshake Protocol

The SSL handshake protocol is one of the defined higher level clients of the SSL record protocol. This protocol is used to negotiate the secure attributes of a session. Handshake messages are supplied to the SSL record layer, where they are encapsulated within one or more SSLPlaintext 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 in the order they must be sent; sending handshake messages in an unexpected order results in a fatal error.

5.6.1. Hello messages

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

5.6.1.1. Hello Request

The hello request message may be sent by the server at any time, but will be ignored by the client if the handshake protocol is already underway. It is a simple notification that the client should begin the negotiation process anew by sending a client hello message when convenient.

Note: Since handshake messages are intended to have transmission precedence over application data, it is expected that the negotiation begin in no more than one or two times the transmission time of a maximum-length application data message.

After sending a hello request, servers should not repeat the request until the subsequent handshake negotiation is complete. A client that receives a hello request while in a handshake negotiation state should simply ignore the message.

The structure of a hello request message is as follows:

```
struct { } HelloRequest;
```

5.6.1.2. Client Hello

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. 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 according to the sender's internal clock. Clocks are not required to be set correctly by the basic SSL 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 simultaneous independent secure connections without repeating the full handshake protocol. The actual contents of the SessionID are defined by the server.

```
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 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 (first choice first). Each CipherSuite defines both a key exchange algorithm and a CipherSpec. 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. If the server supports none of those specified by the client, the session must fail.

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

Issue: Which compression methods to support is under investigation.

The structure of the client hello is as follows.

```
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 SSL protocol by which the client wishes to communicate during this session. This should be the most recent (highest valued) version supported by the client. For this version of the specification, the version will be 3.0 (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, sorted 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.6.

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), this vector must include at least the compression_method from that session. All implementations must support CompressionMethod.null.

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.

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

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.

5.6.1.3. Server Hello

The server processes the client hello message and responds with either a handshake_failure alert or server hello 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 will be 3.0 (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.

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.

5.6.2. Server Certificate

If the server is to be authenticated (which is generally the case), the server sends its certificate immediately following the server hello message. The certificate type must be appropriate for the selected cipher suite's key exchange algorithm, and is generally an X.509.v3 certificate (or a modified X.509 certificate in the case of FORTEZZA(tm) [FOR]). The same message type will be used for the client's response to a certificate request message.

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

certificate_list: This is a sequence (chain) of X.509.v3 certificates, ordered with the sender's certificate first followed by any certificate authority certificates proceeding sequentially upward.

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.

5.6.3. Server Key Exchange Message

The server key exchange message is sent by the server if it has no certificate, has a certificate only used for signing (e.g., DSS [DSS] certificates, signing-only RSA [RSA] certificates), or FORTEZZA KEA key exchange is used. This message is not used if the server certificate contains Diffie-Hellman [DH1] parameters.

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, larger RSA keys may be used as signature-only certificates to sign temporary shorter RSA keys for key exchange.

```
enum { rsa, diffie_hellman, fortezza_kea }
       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 (gX mod p).

```
struct {
    opaque r_s [128];
} ServerFortezzaParams;
```

r_s: Server random number for FORTEZZA KEA (Key Exchange Algorithm).

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

params: The server's key exchange parameters.

signed_params: 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;

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

5.6.4. Certificate Request

A non-anonymous server can optionally request a certificate from the client, if appropriate for the selected cipher suite.

```
     enum {
         rsa_sign(1), dss_sign(2), rsa_fixed_dh(3), dss_fixed_dh(4),
         rsa_ephemeral_dh(5), dss_ephemeral_dh(6), fortezza_kea(20),
         (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.

Note: DistinguishedName is derived from [X509].

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

5.6.5. Server Hello Done

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.

```
struct { } ServerHelloDone;
```

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.

5.6.6. Client Certificate

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 no_certificate alert instead. This alert is only a warning; however, the server may respond with a fatal handshake failure alert if client authentication is required. Client certificates are sent using the certificate defined in Section 5.6.2.

Note: Client Diffie-Hellman certificates must match the server specified Diffie-Hellman parameters.

5.6.7. Client Key Exchange Message

The choice of messages depends on which public key algorithm(s) has (have) been selected. See Section 5.6.3 for the KeyExchangeAlgorithm definition.

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

The information to select the appropriate record structure is in the pending session state (see Section 5.1).

5.6.7.1. RSA Encrypted Premaster Secret Message

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

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 6.1.

5.6.7.2. FORTEZZA Key Exchange Message

Under FORTEZZA, the client derives a token encryption key (TEK) using the FORTEZZA 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 IVs 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 y_signature[40];
          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_signature: y_signature is the signature of the KEA public key, signed with the client's DSS private key.

y_c: The client's Yc value (public key) for the KEA calculation. If the client has sent 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 a certificate without a suitable public key, y_c is used and y_signature is the KEA public key signed with the client's DSS private key. 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: The IV for the client write key.

server_write_iv: The IV for the server write key.

master_secret_iv: This is the IV for the TEK used to encrypt the premaster secret.

pre_master_secret: A random value, generated by the client and used to generate the master secret, as specified in Section 6.1. In the above structure, it is encrypted using the TEK.

5.6.7.3. Client Diffie-Hellman Public Value

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

```
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).

5.6.8. Certificate Verify

This message is used to provide explicit verification of a client certificate. This message is only sent following any client certificate that has signing capability (i.e., all certificates except those containing fixed Diffie-Hellman parameters).

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

```
CertificateVerify.signature.md5_hash
           MD5(master_secret + pad_2 +
               MD5(handshake_messages + master_secret + pad_1));
Certificate.signature.sha_hash
           SHA(master_secret + pad_2 +
               SHA(handshake_messages + master_secret + pad_1));
```

pad_1: This is identical to the pad_1 defined in Section 5.2.3.1.

pad_2: This is identical to the pad_2 defined in Section 5.2.3.1.

Here, handshake_messages refers to all handshake messages starting at client hello up to but not including this message.

5.6.9. Finished

A finished message is always sent immediately after a change cipher spec message to verify that the key exchange and authentication processes were successful. The finished message is the first protected with the just-negotiated algorithms, keys, and secrets. No acknowledgment of the finished message is required; parties may begin sending encrypted data immediately after sending the finished message. Recipients of finished messages must verify that the contents are correct.

```
enum { client(0x434C4E54), server(0x53525652) } Sender;

struct {
    opaque md5_hash[16];
    opaque sha_hash[20];
} Finished;
```

md5_hash: MD5(master_secret + pad2 + MD5(handshake_messages + Sender + master_secret + pad1));

sha_hash: SHA(master_secret + pad2 + SHA(handshake_messages + Sender + master_secret + pad1));

handshake_messages: All of the data from all handshake messages up to but not including this message. This is only data visible at the handshake layer and does not include record layer headers.

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

The hash contained in finished messages sent by the server incorporate Sender.server; those sent by the client incorporate Sender.client. The value handshake_messages includes all handshake messages starting at client hello up to but not including this finished message. This may be different from handshake_messages in Section 5.6.8 because it would include the certificate verify message (if sent).

Note: Change cipher spec messages are not handshake messages and are not included in the hash computations.

5.7. Application Data Protocol

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

6. Cryptographic Computations

The key exchange, authentication, encryption, and MAC algorithms are determined by the cipher_suite selected by the server and revealed in the server hello message.

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 =
  MD5(pre_master_secret + SHA('A' + pre_master_secret +
      ClientHello.random + ServerHello.random)) +
  MD5(pre_master_secret + SHA('BB' + pre_master_secret +
      ClientHello.random + ServerHello.random)) +
  MD5(pre_master_secret + SHA('CCC' + pre_master_secret +
      ClientHello.random + ServerHello.random));
```

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 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 SSL 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 and MAC algorithms are set to SSL_NULL_WITH_NULL_NULL at the beginning of the SSL 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 hashed into a sequence of secure bytes, which are assigned to the MAC secrets, keys, and non-export IVs required by the current CipherSpec (see Appendix A.7). 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:

```
opaque MasterSecret[48]
ClientHello.random
ServerHello.random
```

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 =
  MD5(master_secret + SHA('A' + master_secret +
                          ServerHello.random +
                          ClientHello.random)) +
  MD5(master_secret + SHA('BB' + master_secret +
                          ServerHello.random +
                          ClientHello.random)) +
  MD5(master_secret + SHA('CCC' + master_secret +
                          ServerHello.random +
                          ClientHello.random)) + [...];
```

until enough output has been generated. Then, the key_block is 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 = MD5(client_write_key +
                             ClientHello.random +
                             ServerHello.random);
final_server_write_key = MD5(server_write_key +
                             ServerHello.random +
                             ClientHello.random);
```

Exportable encryption algorithms derive their IVs from the random messages:

```
client_write_IV = MD5(ClientHello.random + ServerHello.random);
server_write_IV = MD5(ServerHello.random + ClientHello.random);
```

MD5 outputs are trimmed to the appropriate size by discarding the least-significant bytes.

6.2.2.1. Export Key Generation Example

SSL_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. MD5 produces 16 bytes of output per call, so three calls to MD5 are required. The MD5 outputs are concatenated into a 48-byte key_block with the first MD5 call providing bytes zero through 15, the second providing bytes 16 through 31, etc. The key_block is partitioned, and the write keys are salted because this is an exportable encryption algorithm.

```
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 = MD5(client_write_key +
                             ClientHello.random +
                             ServerHello.random)[0..15];
final_server_write_key = MD5(server_write_key +
                             ServerHello.random +
                             ClientHello.random)[0..15];
client_write_IV = MD5(ClientHello.random +
                      ServerHello.random)[0..7];
server_write_IV = MD5(ServerHello.random +
                      ClientHello.random)[0..7];
```

7. Security Considerations

See Appendix F.

8. Informative References

[DH1] Diffie, W. and M. Hellman, "New Directions in Cryptography", IEEE Transactions on Information Theory V. IT-22, n. 6, pp. 74-84, June 1977.

[SSL-2] Hickman, K., "The SSL Protocol", February 1995.

[3DES] Tuchman, W., "Hellman Presents No Shortcut Solutions To DES", IEEE Spectrum, v. 16, n. 7, pp 40-41, July 1979.

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

[DSS] NIST FIPS PUB 186, "Digital Signature Standard", National Institute of Standards and Technology U.S. Department of Commerce, May 1994.

[FOR] NSA X22, "FORTEZZA: Application Implementers Guide", Document # PD4002103-1.01, April 1995.

[RFC0959] Postel, J. and J. Reynolds, "File Transfer Protocol", STD 9, RFC 959, October 1985.

[RFC0791] Postel, J., "Internet Protocol", STD 5, RFC 791, September 1981.

[RFC1945] Berners-Lee, T., Fielding, R., and H. Nielsen, "Hypertext Transfer Protocol -- HTTP/1.0", RFC 1945, May 1996.

[RFC1321] Rivest, R., "The MD5 Message-Digest Algorithm", RFC 1321, April 1992.

[RFC0793] Postel, J., "Transmission Control Protocol", STD 7, RFC 793, September 1981.

[RFC0854] Postel, J. and J. Reynolds, "Telnet Protocol Specification", STD 8, RFC 854, May 1983.

[RFC1832] Srinivasan, R., "XDR: External Data Representation Standard", RFC 1832, August 1995.

[RFC2104] Krawczyk, H., Bellare, M., and R. Canetti, "HMAC: Keyed-Hashing for Message Authentication", RFC 2104, February 1997.

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

[PKCS1] RSA Laboratories, "PKCS #1: RSA Encryption Standard version 1.5", November 1993.

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

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

[RSA] Rivest, R., Shamir, A., and L. Adleman, "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems", Communications of the ACM v. 21, n. 2 pp. 120-126., February 1978.

[SCH] Schneier, B., "Applied Cryptography: Protocols, Algorithms, and Source Code in C", John Wiley & Sons, 1994.

[SHA] NIST FIPS PUB 180-1, "Secure Hash Standard", May 1994.

National Institute of Standards and Technology, U.S. Department of Commerce, DRAFT

[X509] CCITT, "The Directory - Authentication Framework", Recommendation X.509 , 1988.

[RSADSI] RSA Data Security, Inc., "Unpublished works".

Appendix A. Protocol Constant Values

This section describes protocol types and constants.

A.1. Record Layer

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

ProtocolVersion version = { 3,0 };

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

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

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

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

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

block-ciphered struct {
    opaque content[SSLCompressed.length];
```

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

A.2. Change Cipher Specs Message

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

A.3. Alert Messages

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

        enum {
            close_notify(0),
            unexpected_message(10),
            bad_record_mac(20),
            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;

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

A.4. Handshake Protocol

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

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

A.4.1. Hello Messages

```
        struct { } HelloRequest;

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

        opaque SessionID<0..32>;

        uint8 CipherSuite[2];

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

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

```
} ClientHello;

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

A.4.2. Server Authentication and Key Exchange Messages

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

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

enum { rsa, diffie_hellman, fortezza_kea } KeyExchangeAlgorithm;

struct {
    opaque RSA_modulus<1..2^16-1>;
    opaque RSA_exponent<1..2^16-1>;
} ServerRSAParams;

struct {
    opaque DH_p<1..2^16-1>;
    opaque DH_g<1..2^16-1>;
    opaque DH_Ys<1..2^16-1>;
} ServerDHParams;

struct {
    opaque r_s [128]
} ServerFortezzaParams

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

```
        enum { anonymous, rsa, dsa } SignatureAlgorithm;

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


        enum {
            RSA_sign(1), DSS_sign(2), RSA_fixed_DH(3),
            DSS_fixed_DH(4), RSA_ephemeral_DH(5), DSS_ephemeral_DH(6),
            FORTEZZA_MISSI(20), (255)
        } CertificateType;

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

        struct {
            CertificateType certificate_types<1..2^8-1>;
            DistinguishedName certificate_authorities<3..2^16-1>;
        } CertificateRequest;

        struct { } ServerHelloDone;
```

A.5. Client Authentication and Key Exchange Messages

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

        struct {
            ProtocolVersion client_version;
            opaque random[46];
        } PreMasterSecret;

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

```
        struct {
            opaque y_c<0..128>;
            opaque r_c[128];
            opaque y_signature[40];
            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];
            opaque encrypted_preMasterSecret[48];
        } FortezzaKeys;

        enum { implicit, explicit } PublicValueEncoding;

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

        struct {
            Signature signature;
        } CertificateVerify;
```

A.5.1. Handshake Finalization Message

```
        struct {
            opaque md5_hash[16];
            opaque sha_hash[20];
        } Finished;
```

A.6. The CipherSuite

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

A CipherSuite defines a cipher specifications supported in SSL version 3.0.

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

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

```
  CipherSuite SSL_RSA_WITH_NULL_MD5                  = { 0x00,0x01 };
  CipherSuite SSL_RSA_WITH_NULL_SHA                  = { 0x00,0x02 };
  CipherSuite SSL_RSA_EXPORT_WITH_RC4_40_MD5         = { 0x00,0x03 };
  CipherSuite SSL_RSA_WITH_RC4_128_MD5               = { 0x00,0x04 };
  CipherSuite SSL_RSA_WITH_RC4_128_SHA               = { 0x00,0x05 };
  CipherSuite SSL_RSA_EXPORT_WITH_RC2_CBC_40_MD5     = { 0x00,0x06 };
  CipherSuite SSL_RSA_WITH_IDEA_CBC_SHA              = { 0x00,0x07 };
  CipherSuite SSL_RSA_EXPORT_WITH_DES40_CBC_SHA      = { 0x00,0x08 };
  CipherSuite SSL_RSA_WITH_DES_CBC_SHA               = { 0x00,0x09 };
  CipherSuite SSL_RSA_WITH_3DES_EDE_CBC_SHA          = { 0x00,0x0A };
```

The following CipherSuite definitions are used for server-authenticated (and optionally client-authenticated) Diffie-Hellman. DH denotes cipher suites in which the server's certificate contains the Diffie-Hellman parameters signed by the certificate authority (CA). DHE denotes ephemeral Diffie-Hellman, where the Diffie-Hellman parameters are signed by a DSS or RSA certificate, which has been signed by the CA. The signing algorithm used is specified after the DH or DHE parameter. In all cases, the client must have the same type of certificate, and must use the Diffie-Hellman parameters chosen by the server.

```
  CipherSuite SSL_DH_DSS_EXPORT_WITH_DES40_CBC_SHA   = { 0x00,0x0B };
  CipherSuite SSL_DH_DSS_WITH_DES_CBC_SHA            = { 0x00,0x0C };
  CipherSuite SSL_DH_DSS_WITH_3DES_EDE_CBC_SHA       = { 0x00,0x0D };
  CipherSuite SSL_DH_RSA_EXPORT_WITH_DES40_CBC_SHA   = { 0x00,0x0E };
  CipherSuite SSL_DH_RSA_WITH_DES_CBC_SHA            = { 0x00,0x0F };
  CipherSuite SSL_DH_RSA_WITH_3DES_EDE_CBC_SHA       = { 0x00,0x10 };
  CipherSuite SSL_DHE_DSS_EXPORT_WITH_DES40_CBC_SHA  = { 0x00,0x11 };
  CipherSuite SSL_DHE_DSS_WITH_DES_CBC_SHA           = { 0x00,0x12 };
  CipherSuite SSL_DHE_DSS_WITH_3DES_EDE_CBC_SHA      = { 0x00,0x13 };
  CipherSuite SSL_DHE_RSA_EXPORT_WITH_DES40_CBC_SHA  = { 0x00,0x14 };
  CipherSuite SSL_DHE_RSA_WITH_DES_CBC_SHA           = { 0x00,0x15 };
  CipherSuite SSL_DHE_RSA_WITH_3DES_EDE_CBC_SHA      = { 0x00,0x16 };
```

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

```
  CipherSuite SSL_DH_anon_EXPORT_WITH_RC4_40_MD5     = { 0x00,0x17 };
  CipherSuite SSL_DH_anon_WITH_RC4_128_MD5           = { 0x00,0x18 };
  CipherSuite SSL_DH_anon_EXPORT_WITH_DES40_CBC_SHA  = { 0x00,0x19 };
  CipherSuite SSL_DH_anon_WITH_DES_CBC_SHA           = { 0x00,0x1A };
  CipherSuite SSL_DH_anon_WITH_3DES_EDE_CBC_SHA      = { 0x00,0x1B };
```

The final cipher suites are for the FORTEZZA token.

```
  CipherSuite SSL_FORTEZZA_KEA_WITH_NULL_SHA         = { 0X00,0X1C };
  CipherSuite SSL_FORTEZZA_KEA_WITH_FORTEZZA_CBC_SHA = { 0x00,0x1D };
  CipherSuite SSL_FORTEZZA_KEA_WITH_RC4_128_SHA      = { 0x00,0x1E };
```

Note: All cipher suites whose first byte is 0xFF are considered private and can be used for defining local/experimental algorithms. Interoperability of such types is a local matter.

A.7. The CipherSpec

A cipher suite identifies a CipherSpec. These structures are part of the SSL session state. The CipherSpec includes:

```
enum { stream, block } CipherType;

enum { true, false } IsExportable;

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

enum { null, md5, sha } MACAlgorithm;

struct {
    BulkCipherAlgorithm bulk_cipher_algorithm;
    MACAlgorithm mac_algorithm;
    CipherType cipher_type;
    IsExportable is_exportable
    uint8 hash_size;
    uint8 key_material;
    uint8 IV_size;
} CipherSpec;
```

Appendix B. Glossary

application protocol: An application protocol is a protocol that normally layers directly on top of the transport layer (e.g., TCP/IP [RFC0793]/[RFC0791]). Examples include HTTP [RFC1945], TELNET [RFC0959], FTP [RFC0854], and SMTP.

asymmetric cipher: See public key cryptography.

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

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

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

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

certificate: As part of the X.509 protocol (a.k.a. ISO Authentication framework), certificates are assigned by a trusted certificate authority and provide verification of a party's identity and may also supply its public key.

client: The application entity that initiates a connection to a server.

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

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

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

Data Encryption Standard (DES): DES is a very widely used symmetric encryption algorithm. DES is a block cipher [DES] [3DES].

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

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

FORTEZZA: A PCMCIA card that provides both encryption and digital signing.

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

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

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

Message Authentication Code (MAC): A Message Authentication Code is a one-way hash computed from a message and some secret data. Its purpose is to detect if the message has been altered.

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

MD5: MD5 [RFC1321] is a secure hashing function that converts an arbitrarily long data stream into a digest of fixed size.

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

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

RC2, RC4: Proprietary bulk ciphers from RSA Data Security, Inc. (There is no good reference to these as they are unpublished works; however, see [RSADSI]). RC2 is a block cipher and RC4 is a stream cipher.

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

salt: Non-secret random data used to make export encryption keys resist precomputation attacks.

server: The server is the application entity that responds to requests for connections from clients. The server is passive, waiting for requests from clients.

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

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

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

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

SHA: The Secure Hash Algorithm is defined in FIPS PUB 180-1. It produces a 20-byte output [SHA].

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

symmetric cipher: See bulk cipher.

Appendix C. CipherSuite Definitions

CipherSuite	Is Exportable	Key Exchange	Cipher	Hash
SSL_NULL_WITH_NULL_NULL	*	NULL	NULL	NULL
SSL_RSA_WITH_NULL_MD5	*	RSA	NULL	MD5
SSL_RSA_WITH_NULL_SHA	*	RSA	NULL	SHA
SSL_RSA_EXPORT_WITH_RC4_40_MD5	*	RSA_EXPORT	RC4_40	MD5
SSL_RSA_WITH_RC4_128_MD5		RSA	RC4_128	MD5
SSL_RSA_WITH_RC4_128_SHA		RSA	RC4_128	SHA
SSL_RSA_EXPORT_WITH_RC2_CBC_40_MD5	*	RSA_EXPORT	RC2_CBC_40	MD5
SSL_RSA_WITH_IDEA_CBC_SHA		RSA	IDEA_CBC	SHA
SSL_RSA_EXPORT_WITH_DES40_CBC_SHA	*	RSA_EXPORT	DES40_CBC	SHA
SSL_RSA_WITH_DES_CBC_SHA		RSA	DES_CBC	SHA
SSL_RSA_WITH_3DES_EDE_CBC_SHA		RSA	3DES_EDE_CBC	SHA
SSL_DH_DSS_EXPORT_WITH_DES40_CBC_SHA	*	DH_DSS_EXPORT	DES40_CBC	SHA
SSL_DH_DSS_WITH_DES_CBC_SHA		DH_DSS	DES_CBC	SHA
SSL_DH_DSS_WITH_3DES_EDE_CBC_SHA		DH_DSS	3DES_EDE_CBC	SHA
SSL_DH_RSA_EXPORT_WITH_DES40_CBC_SHA	*	DH_RSA_EXPORT	DES40_CBC	SHA
SSL_DH_RSA_WITH_DES_CBC_SHA		DH_RSA	DES_CBC	SHA
SSL_DH_RSA_WITH_3DES_EDE_CBC_SHA		DH_RSA	3DES_EDE_CBC	SHA
SSL_DHE_DSS_EXPORT_WITH_DES40_CBC_SHA	*	DHE_DSS_EXPORT	DES40_CBC	SHA
SSL_DHE_DSS_WITH_DES_CBC_SHA		DHE_DSS	DES_CBC	SHA
SSL_DHE_DSS_WITH_3DES_EDE_CBC_SHA		DHE_DSS	3DES_EDE_CBC	SHA
SSL_DHE_RSA_EXPORT_WITH_DES40_CBC_SHA	*	DHE_RSA_EXPORT	DES40_CBC	SHA
SSL_DHE_RSA_WITH_DES_CBC_SHA		DHE_RSA	DES_CBC	SHA
SSL_DHE_RSA_WITH_3DES_EDE_CBC_SHA		DHE_RSA	3DES_EDE_CBC	SHA
SSL_DH_anon_EXPORT_WITH_RC4_40_MD5	*	DH_anon_EXPORT	RC4_40	MD5
SSL_DH_anon_WITH_RC4_128_MD5		DH_anon	RC4_128	MD5
SSL_DH_anon_EXPORT_WITH_DES40_CBC_SHA		DH_anon	DES40_CBC	SHA
SSL_DH_anon_WITH_DES_CBC_SHA		DH_anon	DES_CBC	SHA
SSL_DH_anon_WITH_3DES_EDE_CBC_SHA		DH_anon	3DES_EDE_CBC	SHA
SSL_FORTEZZA_KEA_WITH_NULL_SHA		FORTEZZA_KEA	NULL	SHA
SSL_FORTEZZA_KEA_WITH_FORTEZZA_CBC_SHA		FORTEZZA_KEA	FORTEZZA_CBC	SHA
SSL_FORTEZZA_KEA_WITH_RC4_128_SHA		FORTEZZA_KEA	RC4_128	SHA

Key Exchange Algorithm	Description	Key Size Limit
DHE_DSS	Ephemeral DH with DSS signatures	None
DHE_DSS_EXPORT	Ephemeral DH with DSS signatures	DH = 512 bits
DHE_RSA	Ephemeral DH with RSA signatures	None
DHE_RSA_EXPORT	Ephemeral DH with RSA signatures	DH = 512 bits, RSA = none
DH_anon	Anonymous DH, no signatures	None
DH_anon_EXPORT	Anonymous DH, no signatures	DH = 512 bits
DH_DSS	DH with DSS-based certificates	None
DH_DSS_EXPORT	DH with DSS-based certificates	DH = 512 bits
DH_RSA	DH with RSA-based certificates	None
DH_RSA_EXPORT	DH with RSA-based certificates	DH = 512 bits, RSA = none
FORTEZZA_KEA	FORTEZZA KEA. Details unpublished	N/A
NULL	No key exchange	N/A
RSA	RSA key exchange	None
RSA_EXPORT	RSA key exchange	RSA = 512 bits

Table 1

Key size limit: The key size limit gives the size of the largest public key that can be legally used for encryption in cipher suites that are exportable.

Cipher	Cipher Type	IsExportable	Key Material	Exp. Key Material	Effective Key Bits	IV Size	Block Size
NULL	Stream	*	0	0	0	0	N/A
FORTEZZA_CBC	Block		NA (**)	12 (**)	96 (**)	20 (**)	8
IDEA_CBC	Block		16	16	128	8	8
RC2_CBC_40	Block	*	5	16	40	8	8
RC4_40	Stream	*	5	16	40	0	N/A
RC4_128	Stream		16	16	128	0	N/A
DES40_CBC	Block	*	5	8	40	8	8
DES_CBC	Block		8	8	56	8	8
3DES_EDE_CBC	Block		24	24	168	8	8

* Indicates IsExportable is true.
** FORTEZZA uses its own key and IV generation algorithms.

Table 2

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

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

Effective Key Bits: How much entropy material is in the key material being fed into the encryption routines.

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

Table 3

Appendix D. Implementation Notes

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

D.1. Temporary RSA Keys

US export restrictions limit RSA keys used for encryption to 512 bits, but do not place any limit on lengths of RSA keys used for signing operations. Certificates often need to be larger than 512 bits, since 512-bit RSA keys are not secure enough for high-value transactions or for applications requiring long-term security. Some certificates are also designated signing-only, in which case they cannot be used for key exchange.

When the public key in the certificate cannot be used for encryption, the server signs a temporary RSA key, which is then exchanged. In exportable applications, the temporary RSA key should be the maximum allowable length (i.e., 512 bits). Because 512-bit RSA keys are relatively insecure, they should be changed often. For typical electronic commerce applications, it is suggested that keys be changed daily or every 500 transactions, and more often if possible. Note that while it is acceptable to use the same temporary key for multiple transactions, it must be signed each time it is used.

RSA key generation is a time-consuming process. In many cases, a low-priority process can be assigned the task of key generation. Whenever a new key is completed, the existing temporary key can be replaced with the new one.

D.2. Random Number Generation and Seeding

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

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

Note: The seeding functions in RSAREF and versions of BSAFE prior to 3.0 are order independent. For example, if 1000 seed bits are supplied, one at a time, in 1000 separate calls to the seed function, the PRNG will end up in a state that depends only on the number of 0 or 1 seed bits in the seed data (i.e., there are 1001 possible final states). Applications using BSAFE or RSAREF must take extra care to ensure proper seeding.

D.3. Certificates and Authentication

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

D.4. CipherSuites

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

D.5. FORTEZZA

This section describes implementation details for cipher suites that make use of the FORTEZZA hardware encryption system.

D.5.1. Notes on Use of FORTEZZA Hardware

A complete explanation of all issues regarding the use of FORTEZZA hardware is outside the scope of this document. However, there are a few special requirements of SSL that deserve mention.

Because SSL is a full duplex protocol, two crypto states must be maintained, one for reading and one for writing. There are also a number of circumstances that can result in the crypto state in the FORTEZZA card being lost. For these reasons, it’s recommended that the current crypto state be saved after processing a record, and loaded before processing the next.

After the client generates the TEK, it also generates two message encryption keys (MEKs), one for reading and one for writing. After generating each of these keys, the client must generate a corresponding IV and then save the crypto state. The client also uses the TEK to generate an IV and encrypt the premaster secret. All three IVs are sent to the server, along with the wrapped keys and the encrypted premaster secret in the client key exchange message. At this point, the TEK is no longer needed, and may be discarded.

On the server side, the server uses the master IV and the TEK to decrypt the premaster secret. It also loads the wrapped MEKs into the card. The server loads both IVs to verify that the IVs match the keys. However, since the card is unable to encrypt after loading an IV, the server must generate a new IV for the server write key. This IV is discarded.

When encrypting the first encrypted record (and only that record), the server adds 8 bytes of random data to the beginning of the fragment. These 8 bytes are discarded by the client after decryption. The purpose of this is to synchronize the state on the client and server resulting from the different IVs.

D.5.2. FORTEZZA Cipher Suites

5) FORTEZZA_NULL_WITH_NULL_SHA: Uses the full FORTEZZA key exchange, including sending server and client write keys and IVs.

D.5.3. FORTEZZA Session Resumption

There are two possibilities for FORTEZZA session restart: 1) Never restart a FORTEZZA session. 2) Restart a session with the previously negotiated keys and IVs.

Never restarting a FORTEZZA session:

Clients who never restart FORTEZZA sessions should never send session IDs that were previously used in a FORTEZZA session as part of the ClientHello. Servers who never restart FORTEZZA sessions should never send a previous session id on the ServerHello if the negotiated session is FORTEZZA.

Restart a session:

You cannot restart FORTEZZA on a session that has never done a complete FORTEZZA key exchange (that is, you cannot restart FORTEZZA if the session was an RSA/RC4 session renegotiated for FORTEZZA). If you wish to restart a FORTEZZA session, you must save the MEKs and

IVs from the initial key exchange for this session and reuse them for any new connections on that session. This is not recommended, but it is possible.

Appendix E. Version 2.0 Backward Compatibility

Version 3.0 clients that support version 2.0 servers must send version 2.0 client hello messages [SSL-2]. Version 3.0 servers should accept either client hello format. The only deviations from the version 2.0 specification are the ability to specify a version with a value of three and the support for more ciphering types in the CipherSpec.

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

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

```
V2CipherSpec SSL_RC4_128_WITH_MD5          = { 0x01,0x00,0x80 };
V2CipherSpec SSL_RC4_128_EXPORT40_WITH_MD5 = { 0x02,0x00,0x80 };
V2CipherSpec SSL_RC2_CBC_128_CBC_WITH_MD5  = { 0x03,0x00,0x80 };
V2CipherSpec SSL_RC2_CBC_128_CBC_EXPORT40_WITH_MD5
                                           = { 0x04,0x00,0x80 };
V2CipherSpec SSL_IDEA_128_CBC_WITH_MD5     = { 0x05,0x00,0x80 };
V2CipherSpec SSL_DES_64_CBC_WITH_MD5       = { 0x06,0x00,0x40 };
V2CipherSpec SSL_DES_192_EDE3_CBC_WITH_MD5 = { 0x07,0x00,0xC0 };
```

Cipher specifications introduced in version 3.0 can be included in version 2.0 client hello messages using the syntax below. Any V2CipherSpec element with its first byte equal to zero will be ignored by version 2.0 servers. Clients sending any of the above V2CipherSpecs should also include the version 3.0 equivalent (see Appendix A.6):

```
V2CipherSpec (see Version 3.0 name) = { 0x00, CipherSuite };
```

E.1. Version 2 Client Hello

The version 2.0 client hello message is presented below using this document's presentation model. The true definition is still assumed to be the SSL version 2.0 specification.

```
      uint8 V2CipherSpec[3];

      struct {
          unit8 msg_type;
          Version version;
          uint16 cipher_spec_length;
          uint16 session_id_length;
          uint16 challenge_length;
          V2CipherSpec cipher_specs[V2ClientHello.cipher_spec_length];
          opaque session_id[V2ClientHello.session_id_length];
          Random challenge;
      } V2ClientHello;
```

session msg_type: This field, in conjunction with the version field, identifies a version 2 client hello message. The value should equal one (1).

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

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

session_id_length: This field must have a value of either zero or 16. If zero, the client is creating a new session. If 16, the session_id field will contain the 16 bytes of session identification.

challenge_length: The length in bytes of the client's challenge to the server to authenticate itself. This value must be 32.

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

session_id: If this field's length is not zero, it will contain the identification for a session that the client wishes to resume.

challenge: The client's challenge to the server for the server to identify itself is a (nearly) arbitrary length random. The version 3.0 server will right justify the challenge data to become the ClientHello.random data (padded with leading zeroes, if necessary), as specified in this version 3.0 protocol. If the length of the challenge is greater than 32 bytes, then only the last 32 bytes are used. It is legitimate (but not necessary) for a V3 server to reject a V2 ClientHello that has fewer than 16 bytes of challenge data.

Note: Requests to resume an SSL 3.0 session should use an SSL 3.0 client hello.

E.2. Avoiding Man-in-the-Middle Version Rollback

When SSL version 3.0 clients fall back to version 2.0 compatibility mode, they use special PKCS #1 block formatting. This is done so that version 3.0 servers will reject version 2.0 sessions with version 3.0-capable clients.

When version 3.0 clients are in version 2.0 compatibility mode, they set the right-hand (least-significant) 8 random bytes of the PKCS padding (not including the terminal null of the padding) for the RSA encryption of the ENCRYPTED-KEY-DATA field of the CLIENT-MASTER-KEY to 0x03 (the other padding bytes are random). After decrypting the ENCRYPTED-KEY-DATA field, servers that support SSL 3.0 should issue an error if these eight padding bytes are 0x03. Version 2.0 servers receiving blocks padded in this manner will proceed normally.

Appendix F. Security Analysis

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

F.1. Handshake Protocol

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

F.1.1. Authentication and Key Exchange

SSL supports three authentication modes: authentication of both parties, server authentication with an unauthenticated client, and total anonymity. Whenever the server is authenticated, the channel should be secure against man-in-the-middle attacks, but completely anonymous sessions are inherently vulnerable to such attacks.

Anonymous servers cannot authenticate clients, since the client signature in the certificate verify message may require a server certificate to bind the signature to a particular server. If the server is authenticated, its certificate message must provide a valid certificate chain leading to an acceptable certificate authority. Similarly, authenticated clients must supply an acceptable certificate to the server. Each party is responsible for verifying that the other's certificate is valid and has not expired or been revoked.

The general goal of the key exchange process is to create a pre_master_secret known to the communicating parties and not to attackers. The pre_master_secret will be used to generate the master_secret (see Section 6.1). The master_secret is required to generate the finished messages, encryption keys, and MAC secrets (see Sections 5.6.9 and 6.2.2). By sending a correct finished message, parties thus prove that they know the correct pre_master_secret.

F.1.1.1. Anonymous Key Exchange

Completely anonymous sessions can be established using RSA, Diffie-Hellman, or FORTEZZA for key exchange. With anonymous RSA, the client encrypts a pre_master_secret with the server's uncertified public key extracted from the server key exchange message. The result is sent in a client key exchange message. Since eavesdroppers do not know the server's private key, it will be infeasible for them to decode the pre_master_secret.

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

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

F.1.1.2. RSA Key Exchange and Authentication

With RSA, key exchange and server authentication are combined. The public key either may be contained in the server's certificate or may be a temporary RSA key sent in a server key exchange message. When temporary RSA keys are used, they are signed by the server's RSA or DSS certificate. The signature includes the current

ClientHello.random, so old signatures and temporary keys cannot be replayed. Servers may use a single temporary RSA key for multiple negotiation sessions.

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

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

When RSA is used for key exchange, clients are authenticated using the certificate verify message (see Section 5.6.8). The client signs a value derived from the master_secret and all preceding handshake messages. These handshake messages include the server certificate, which binds the signature to the server, and ServerHello.random, which binds the signature to the current handshake process.

F.1.1.3. Diffie-Hellman Key Exchange with Authentication

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

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

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

F.1.1.4. FORTEZZA

FORTEZZA's design is classified, but at the protocol level it is similar to Diffie-Hellman with fixed public values contained in certificates. The result of the key exchange process is the token encryption key (TEK), which is used to wrap data encryption keys, client write key, server write key, and master secret encryption key. The data encryption keys are not derived from the pre_master_secret because unwrapped keys are not accessible outside the token. The encrypted pre_master_secret is sent to the server in a client key exchange message.

F.1.2. Version Rollback Attacks

Because SSL version 3.0 includes substantial improvements over SSL version 2.0, attackers may try to make version 3.0-capable clients and servers fall back to version 2.0. This attack is occurring if (and only if) two version 3.0-capable parties use an SSL 2.0 handshake.

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

F.1.3. Detecting Attacks against the Handshake Protocol

An attacker might try to influence the handshake exchange to make the parties select different encryption algorithms than they would normally choose. Because many implementations will support 40-bit exportable encryption and some may even support null encryption or MAC algorithms, this attack is of particular concern.

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

F.1.4. Resuming Sessions

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

Sessions cannot be resumed unless both the client and server agree. If either party suspects that the session may have been compromised, or that certificates may have expired or been revoked, it should force a full handshake. An upper limit of 24 hours is suggested for session ID lifetimes, since an attacker who obtains a master_secret may be able to impersonate the compromised party until the corresponding session ID is retired. Applications that may be run in relatively insecure environments should not write session IDs to stable storage.

F.1.5. MD5 and SHA

SSL uses hash functions very conservatively. Where possible, both MD5 and SHA are used in tandem to ensure that non-catastrophic flaws in one algorithm will not break the overall protocol.

F.2. Protecting Application Data

The master_secret is hashed with the ClientHello.random and ServerHello.random to produce unique data encryption keys and MAC secrets for each connection. FORTEZZA encryption keys are generated by the token, and are not derived from the master_secret.

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

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

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

F.3. Final Notes

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

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

Appendix G. Acknowledgements

G.1. Other Contributors

Martin Abadi
Digital Equipment Corporation
ma@pa.dec.com

Robert Relyea
Netscape Communications
relyea@netscape.com

Taher Elgamal
Netscape Communications
elgamal@netscape.com

Jim Roskind
Netscape Communications
jar@netscape.com

Anil Gangolli
Netscape Communications
gangolli@netscape.com

Micheal J. Sabin, Ph.D.
Consulting Engineer
msabin@netcom.com

Kipp E.B. Hickman
Netscape Communications
kipp@netscape.com

Tom Weinstein
Netscape Communications
tomw@netscape.com

G.2. Early Reviewers

Robert Baldwin
RSA Data Security, Inc.
baldwin@rsa.com

Clyde Monma
Bellcore
clyde@bellcore.com

George Cox
Intel Corporation
cox@ibeam.jf.intel.com

Eric Murray
ericm@lne.com

Cheri Dowell
Sun Microsystems
cheri@eng.sun.com

Avi Rubin
Bellcore
rubin@bellcore.com

Stuart Haber
Bellcore
stuart@bellcore.com

Don Stephenson
Sun Microsystems
don.stephenson@eng.sun.com

Burt Kaliski
RSA Data Security, Inc.
burt@rsa.com

Joe Tardo
General Magic
tardo@genmagic.com

Authors' Addresses

Alan O. Freier
Netscape Communications

Philip Karlton
Netscape Communications

Paul C. Kocher
Independent Consultant