TinySSL Protocol Specification

Revision 1

COSI 107a

April 2023

1 Status of this memo

TinySSL is a simplified version of the original published Secure Sockets Layer (SSL) protocol, which was labeled as version 2. It is intended for use as an exercise in implementing the protocol, both to understand some essential ideas in its design, as well as to gain experience implementing a network protocol.

The protocol has several flaws, and its successor Transport Layer Security (TLS) has fixed many of them through successive versions. This version is simpler to implement for practice in understanding some of the key ideas and programming techniques.

Protocols for the Internet are traditionally called "Requests for Comments", or "RFCs", dating back to early days of people circulating ideas for how things on the network should work. Over time, it became necessary to identify some documents as standards and formalize other parts of the standardization proces. The idea of an "Internet Draft" became common as a way to share something with the community before it was ready to start serious discussion as a possible standard.

This memo is based on "The SSL Protocol", published in April 1995 by Kipp Hickman, who was then working for Netscape Communications Corp., the vendor of the first commercial web browser and server. It has been adapted for this assignment, while still retaining essential aspects of an Internet protocol specification. You might find it interesting to look at the original linked above as well as the current TLS specifcation at RFC 8446: The Transport Layer Security (TLS) Protocol Version 1.3.

The protocol specified here is for educational purposes only, and it has known security flaws.

2 Introduction

For an introduction to SSL version 2, see draft-hickman-netscape-ssl-00.pdf.

3 Protocol definition

Note: the order of sections in the lab has been changed from the original specification to present the concepts more clearly.

In this protocol description, the type `char` is to be considered to be exactly one byte.

3.1 Typical Protocol Message Flow

The following sequences define several typical protocol message flows for the TinySSL Handshake Protocol. In these examples we have two principals in the conversation: the client and the server. We use a notation commonly found in the literature [10]. When something is enclosed in curly braces "{something}key" then the something has been encrypted using "key".

In this lab, we are implementing the protocol without an session identifier and without using client certificates for authentication. Session identifiers reduce overhead and simplify continuing communication between a client and server; client authentication for SSL/TLS has never become common.

The typical message flow without a pre-existing session looks like this:

client-hello	`C -> S: challenge, cipher_specs`
server-hello	`S -> C: connection-id, server_certificate,cipher_specs`
client-master-key	`C -> S: {master_key}server_public_key`
client-finish	`C -> S: {connection-id}client_write_key`
server-verify	`S -> C: {challenge}server_write_key`
server-finish	`S -> C: {new_session_id}server_write_key`

3.2 TinySSL Record Header Format

In TinySSL, all data sent is encapsulated in a record, an object which is composed of a header and some non-zero amount of data. Each record header contains a two or three byte length code. If the most significant bit is set in the first byte of the record length code then the record has no padding and the total header length will be 2 bytes, otherwise the record has padding and the total header length will be 3 bytes. The record header is transmitted before the data portion of the record.

Note that in the long header case (3 bytes total), the second most significant bit in the first byte has special meaning. When zero, the record being sent is a data record. When one, the record being sent is a security escape (Note: SSL v2 did not define any security escapes, reserving them for future versions of the protocol. TinySSL uses escapes as defined below for providing a testing mechanism). In either case, the length code describes how much data is in the record.

The record length code does not include the number of bytes consumed by the record header (2 or 3). For the 2 byte header, the record length is computed by (using a "C"-like notation):

```
RECORD-LENGTH = ((byte[0] & 0x7f) << 8)) | byte[1];
```

Where byte[0] represents the first byte received and byte[1] the second byte received. When the 3 byte header is used, the record length is computed as follows (using a "C"-like notation):

```
RECORD-LENGTH = ((byte[0] & 0x3f) << 8)) | byte[1];
IS-ESCAPE = (byte[0] & 0x40) != 0;
PADDING = byte[2];
```

The record header defines a value called `PADDING`. The `PADDING` value specifies how many bytes of data were appended to the original record by the sender. The padding data is used to make the record length be a multiple of the block ciphers block size when a block cipher is used for encryption.

If `IS-ESCAPE` is non-zero, the data is the record is a diagnostic message sent by the server to help in diagnosing client implementation problems. In some cases, the diagnostic will start with the string "echo:" followed by the data sent by the client. The echoed data is returned unencrypted so that the client can verify that its messages were successfully decrypted. Echo data may be binary or ASCII as appropriate. Other diagnostic messages are almost always ASCII. In a security escape, the value of `PADDING` is always 0.

TinySSL does not use block ciphers or padding, so `PADDING` should always be zero.

3.2.1 TinySSL Record Data Format

The data portion of an TinySSL record is composed of three components (transmitted and received in the order shown):

```
MAC-DATA[MAC-SIZE]
ACTUAL-DATA[N]
PADDING-DATA[PADDING]
```

ACTUAL-DATA is the actual data being transmitted (the message payload). PADDING-DATA is the padding data sent when a block cipher is used and padding is needed. Finally, MAC-DATA is the "Message Authentication Code".

When TinySSL records are sent in the clear, no cipher is used. Consequently the amount of PADDING-DATA will be zero and the amount of MAC-DATA will be zero. When encryption is in effect, the PADDING-DATA will be a function of the cipher block size. The MAC-DATA is a function of the CIPHER-CHOICE (more about that later).

The MAC-DATA is computed as follows:

```
MAC-DATA = HASH[ SECRET, ACTUAL-DATA, PADDING-DATA,SEQUENCE-NUMBER
]
```

Where the SECRET data is fed to the hash function first, followed by the ACTUAL-DATA, which is followed by the PADDING-DATA which is finally followed by the SEQUENCE-NUMBER. The SEQUENCE-NUMBER is a 32 bit value which is presented to the hash function as four bytes, with the first byte being the most significant byte of the sequence number, the second byte being the next most significant byte of the sequence number, the third byte being the third most significant byte, and the fourth byte being the least significant byte (that is, in network byte order or "big endian" order).

MAC-SIZE is a function of the digest algorithm being used. For SHA256, the MAC-SIZE is 32 bytes (256 bits).

The SECRET value is a function of which party is sending the message. If the client is sending the message then the SECRET is the CLIENT-WRITE-KEY (the server will use the SERVER-READ-KEY to verify the MAC). If the client is receiving the message then the SECRET is the CLIENT-READ-KEY (the server will use the SERVER-WRITE-KEY to generate the MAC).

The SEQUENCE-NUMBER is a counter which is incremented by both the sender and the receiver. For each transmission direction, a pair of counters is kept (one by the sender, one by the receiver). Every time a message is sent by a sender the counter is incremented. Sequence numbers are 32 bit unsigned quantities and must wrap to zero after incrementing past 0xFFFFFFFF.

The receiver of a message uses the expected value of the sequence number as input into the MAC HASH function (the HASH function is chosen from the CIPHER-CHOICE). The computed MAC-DATA must agree bit for bit with the transmitted MAC-DATA. If the comparison is not identity then the record is considered damaged, and it is to be treated as if an "I/O Error" had occurred

(i.e. an unrecoverable error is asserted and the connection is closed).

A final consistency check is done when a block cipher is used and the protocol is using encryption. The amount of data present in a record (`RECORD-LENGTH`) must be a multiple of the cipher's block size. If the received record is not a multiple of the cipher's block size then the record is considered damaged, and it is to be treated as if an "I/O Error" had occurred (i.e. an unrecoverable error is asserted and the connection is closed).

The TinySSL Record Layer is used for all TinySSL communications, including handshake messages, security escapes and application data transfers. The TinySSL Record Layer is used by both the client and the server at all times.

For a two byte header, the maximum record length is 32,767 bytes. For the three-byte header, the maximum record length is 16,383 bytes. The TinySSL Handshake Protocol messages are constrained to fit in a single TinySSL Record Protocol record. Application protocol messages are allowed to consume multiple TinySSL Record Protocol records.

Before the first record is sent using TinySSL all sequence numbers are initialized to zero. The transmit sequence number is incremented after every message sent, starting with the `CLIENT-HELLO` and `SERVER-HELLO` messages.

3.3 TinySSL Handshake Protocol Flow

The TinySSL Handshake Protocol is used to negotiate security enhancements to data sent using the TinySSL Record Protocol. The security enhancements consist of authentication, symmetric encryption, and message integrity.

The TinySSL Handshake Protocol has two major phases. The first phase is used to establish private communications. The second phase is used for client authentication (which TinySSL does not include.

3.3.1 Phase 1

The first phase is the initial connection phase where both parties communicate their "hello" messages. The client initiates the conversation by sending the `CLIENT-HELLO` message. The server receives the `CLIENT-HELLO` message and processes it responding with the `SERVER-HELLO` message.

At this point both the client and server have enough information to know whether or not a new "master key" is needed (The master key is used for production of the symmetric encryption session keys). When a new master key is not needed, both the client and the server proceed immediately to phase 2.

When a new master key is needed, the SERVER-HELLO message will contain enough information for the client to generate it. This includes the server's signed certificate (more about that later), a list of bulk cipher specifications (see below), and a connection-id (a connection-id is a randomly generated value generated by the server that is used by the client and server during a single connection). The client generates the master key and responds with a CLIENT-MASTER-KEY message (or an ERROR message if the server information indicates that the client and server cannot agree on a bulk cipher).

It should be noted here that each TinySSL endpoint uses a pair of ciphers per connection (for a total of four ciphers). At each endpoint, one cipher is used for outgoing communications, and one is used for incoming communications. When the client or server generate a session key, they actually generate two keys, the SERVER-READ-KEY (also known as the CLIENT-WRITE-KEY) and the SERVER-WRITE-KEY (also known as the CLIENT-READ-KEY). The master key is used by the client and server to generate the various session keys (more about that later).

Finally, the server sends a SERVER-VERIFY message to the client after the master key has been determined. This final step authenticates the server, because only a server which has the appropriate public key can know the master key.

3.3.2 Phase 2

In original SSL, the second phase is for client authentication. If the client receives a REQUEST-CERTIFICATE message from the server, it should respond with an appropriate ERROR message.

When a party is done authenticating the other party, it sends its finished message. For the client, the CLIENT-FINISHED message contains the encrypted form of the CONNECTION-ID for the server to verify. If the verification fails, the server sends an ERROR message.

Once a party has sent its finished message it must continue to listen to its peers messages until it too receives a finished message. Once a party has both sent a finished message and received its peers finished message, the TinySSL handshake protocol is done. At this point the application protocol begins to operate (Note: the application protocol continues to be layered on the TinySSL Record Protocol).

3.4 Errors

Error handling in the TinySSL connection protocol is very simple. When an error is detected, the detecting party sends a message to the other party. Errors that are not recoverable cause the client and server to abort the secure connection. Servers and client are required to "forget" any session-identifiers associated with a failing connection.

The TinySSL Handshake Protocol defines the following errors:

`NO-CIPHER-ERROR` This error is returned by the client to the server when it cannot find a cipher or key size that it supports that is also supported by the server. This error is not recoverable.

`NO-CERTIFICATE-ERROR` When a `REQUEST-CERTIFICATE` message is sent, this error may be returned if the client has no certificate to reply with. This error is recoverable (for client authentication only).

`BAD-CERTIFICATE-ERROR` This error is returned when a certificate is deemed bad by the receiving party. Bad means that either the signature of the certificate was bad or that the values in the certificate were inappropriate (e.g. a name in the certificate did not match the expected name). This error is recoverable (for client authentication only).

`UNSUPPORTED-CERTIFICATE-TYPE-ERROR` This error is returned when a client/server receives a certificate type that it can't support. This error is recoverable (for client authentication only).

3.5 TinySSL Handshake Protocol Messages

The TinySSL Handshake Protocol messages are encapsulated using the TinySSL Record Protocol and are composed of two parts: a single byte message type code, and some data. The client and server exchange messages until both ends have sent their "finished" message, indicating that they are satisfied with the TinySSL Handshake Protocol (TSSLHP) conversation. While one end may be finished, the other may not, therefore the finished end must continue to receive TinySSL Handshake Protocol messages until it receives a "finished" message from its peer.

After the pair of session keys has been determined by each party, the message bodies are encrypted. For the client, this happens after it verifies the session-identifier or creates a new master key and has sent it to the server. For the server, this happens after the session-identifier is found to be good, or the server receives the client's master key message.

The following notation is used for TSSLHP messages:

```
char MSG-EXAMPLE
char FIELD1
char FIELD2
char THING-MSB
char THING-LSB
char THING-DATA[(MSB<<8)|LSB];
...
```

This notation defines the data in the protocol message, including the message type code. The order is presented top to bottom, with the top most element being transmitted first, and the bottom most element transferred last.

For a number that takes more than one byte to represent it, we need to define what order the bytes come in for things like network protocols and CPU architecture. In TinyTLS, we use 16-bit values for some things. The "most significant byte" (MSB) is the "larger" part, and the "least significant byte" (LSB) is the smaller part. For example, suppose we need to send the number 8,231. In hex, that's 0x2027. Two hex digits represents one byte, so 0x20 would the MSB and 0x27 would be the LSB.

For the `THING-DATA` entry, the MSB and LSB values are actually `THING-MSB` and `THING-LSB` (respectively) and define the number of bytes of data actually present in the message. For example, if `THING-MSB` were zero and `THING-LSB` were 8 then the `THING-DATA` array would be exactly 8 bytes long. This shorthand is used below.

Length codes are unsigned values, and when the MSB and LSB are combined the result is an unsigned value. Unless otherwise specified lengths values are "length in bytes".

3.5.1 Client Only Protocol Messages

There are several messages that are only generated by clients. These messages are never generated by correctly functioning servers. A client receiving such a message closes the connection to the server and returns an error status to the application through some unspecified mechanism.

1. `CLIENT-HELLO` (Phase 1; Sent in the clear)

   ```
   char MSG-CLIENT-HELLO
   ```

```
char CLIENT-VERSION-MSB
char CLIENT-VERSION-LSB
char CIPHER-SPECS-LENGTH-MSB
char CIPHER-SPECS-LENGTH-LSB
char SESSION-ID-LENGTH-MSB
char SESSION-ID-LENGTH-LSB
char CHALLENGE-LENGTH-MSB
char CHALLENGE-LENGTH-LSB
char CIPHER-SPECS-DATA[(MSB<<8)|LSB]
char SESSION-ID-DATA[(MSB<<8)|LSB]
char CHALLENGE-DATA[(MSB<<8)|LSB]
```

When a client first connects to a server it is required to send the `CLIENT-HELLO` message. The server is expecting this message from the client as its first message. It is an error for a client to send anything else as its first message.

The client sends to the server its SSL version, its cipher specs (see below), some challenge data, and the session-identifier data. The session-identifier data is only sent if the client found a session-identifier in its cache for the server, and the `SESSION-ID-LENGTH` will be non-zero. When there is no session-identifier for the server `SESSION-ID-LENGTH` must be zero. The challenge data is used to authenticate the server. After the client and server agree on a pair of session keys, the server returns a `SERVER-VERIFY` message with the encrypted form of the `CHALLENGE-DATA`.

Also note that the server will not send its `SERVER-HELLO` message until it has received the `CLIENT-HELLO` message. This is done so that the server can indicate the status of the client's session-identifier back to the client in the server's first message (i.e. to increase protocol efficiency and reduce the number of round trips required).

The server examines the `CLIENT-HELLO` message and will verify that it can support the client version and one of the client cipher specs. The server can optionally edit the cipher specs, removing any entries it doesn't choose to support. The edited version will be returned in the `SERVER-HELLO` message if the session-identifier is not in the server's cache.

The `CIPHER-SPECS-LENGTH` must be greater than zero and a multiple of 3. The `SESSION-ID-LENGTH` must either be zero or 16. The

CHALLENGE-LENGTH must be greater than or equal to 16 and less than or equal to 32.

This message must be the first message sent by the client to the server. After the message is sent the client waits for a SERVER-HELLO message. Any other message returned by the server (other than ERROR) is disallowed.

2. CLIENT-MASTER-KEY (Phase 1; Sent primarily in the clear)

```
char MSG-CLIENT-MASTER-KEY
char CIPHER-KIND[3]
char ENCRYPTED-KEY-LENGTH-MSB
char ENCRYPTED-KEY-LENGTH-LSB
char KEY-ARG-LENGTH-MSB
char KEY-ARG-LENGTH-LSB
char CLEAR-KEY-DATA[MSB<<8|LSB]
char ENCRYPTED-KEY-DATA[MSB<<8|LSB]
char KEY-ARG-DATA[MSB<<8|LSB]
```

The client sends this message when it has determined a master key for the server to use. Note that when a session-identifier has been agreed upon, this message is not sent.

The CIPHER-KIND field indicates which cipher was chosen from the server's CIPHER-SPECS.

The ENCRYPTED-KEY-DATA contains the secret portions of the MASTER-KEY, encrypted using the server's public key. The encryption block is formatted using block type 2 from PKCS#1 [5]. The data portion of the block is formatted as follows:

```
char SECRET-KEY-DATA[SECRET-LENGTH]
```

SECRET-LENGTH is the number of bytes of each session key that is being transmitted encrypted. The SECRET-LENGTH plus the CLEAR-KEY-LENGTH equals the number of bytes present in the cipher key (as defined by the CIPHER-KIND). It is an error if the SECRET-LENGTH found after decrypting the PKCS#1 formatted encryption block doesn't match the expected value. It is also an error if CLEAR-KEY-LENGTH is non-zero and the CIPHER-KIND is not an export cipher.

If the key algorithm needs an argument (for example, DES-CBC's initialization vector) then the KEY-ARG-LENGTH fields will be non-zero and the KEY-ARG-DATA will contain the relevant data. For the SSL_CK_RC2_128_CBC_WITH_MD5, SSL_CK_RC2_128_CBC_EXPORT40_WITH_MD5, SSL_CK_IDEA_128_CBC_WITH_MD5, SSL_CK_DES_64_CBC_WITH_MD5, and SSL_CK_DES_192_EDE3_CBC_WITH_MD5 algorithms the KEY-ARG data must be present and be exactly 8 bytes long.

Client and server session key production is a function of the CIPHER-CHOICE:

```
SSL_CK_RC4_128_WITH_MD5
SSL_CK_RC4_128_EXPORT40_WITH_MD5
SSL_CK_RC2_128_CBC_WITH_MD5
SSL_CK_RC2_128_CBC_EXPORT40_WITH_MD5
SSL_CK_IDEA_128_CBC_WITH_MD5

KEY-MATERIAL-0 = MD5[ MASTER-KEY, "0", CHALLENGE, CONNECTION-ID ]
KEY-MATERIAL-1 = MD5[ MASTER-KEY, "1", CHALLENGE, CONNECTION-ID ]

CLIENT-READ-KEY = KEY-MATERIAL-0[0-15]
CLIENT-WRITE-KEY = KEY-MATERIAL-1[0-15]
```

Where KEY-MATERIAL-~0[0-15] means the first 16 bytes of the KEY-MATERIAL-~0 data, with KEY-MATERIAL-~0[0] becoming the most significant byte of the CLIENT-READ-KEY.

Data is fed to the MD5 hash function in the order shown, from left to right: first the MASTER-KEY, then the "0" or "1", then the CHALLENGE and then finally the CONNECTION-ID.

Note that the "0" means the ASCII zero character (0x30), not a zero value. "1" means the ASCII 1 character (0x31). MD5 produces 128 bits of output data which are used directly as the key to the cipher algorithm (The most significant byte of the MD5 output becomes the most significant byte of the key material).

```
SSL_CK_DES_64_CBC_WITH_MD5

KEY-MATERIAL-0 = MD5[ MASTER-KEY, CHALLENGE, CONNECTION-ID ]

CLIENT-READ-KEY = KEY-MATERIAL-0[0-7]
CLIENT-WRITE-KEY = KEY-MATERIAL-0[8-15]
```

For DES-CBC, a single 16 bytes of key material are produced using MD5. The first 8 bytes of the MD5 digest are used as the CLIENT-READ-KEY while the remaining 8 bytes are used as the CLIENT-WRITE-KEY. The initialization vector is provided in the KEY-ARG-DATA. Note that the raw key data is not parity adjusted and that this step must be performed before the keys are legitimate DES keys.

```
SSL_CK_DES_192_EDE3_CBC_WITH_MD5
KEY-MATERIAL-0 = MD5[ MASTER-KEY, "0", CHALLENGE, CONNECTION-ID ]
KEY-MATERIAL-1 = MD5[ MASTER-KEY, "1", CHALLENGE, CONNECTION-ID ]
KEY-MATERIAL-2 = MD5[ MASTER-KEY, "2", CHALLENGE, CONNECTION-ID ]

CLIENT-READ-KEY-0 = KEY-MATERIAL-0[0-7]
CLIENT-READ-KEY-1 = KEY-MATERIAL-0[8-15]
CLIENT-READ-KEY-2 = KEY-MATERIAL-1[0-7]
CLIENT-WRITE-KEY-0 = KEY-MATERIAL-1[8-15]
CLIENT-WRITE-KEY-1 = KEY-MATERIAL-2[0-7]
CLIENT-WRITE-KEY-2 = KEY-MATERIAL-2[8-15]
```

Data is fed to the MD5 hash function in the order shown, from left to right: first the MASTER-KEY, then the "0", "1" or "2", then the CHALLENGE and then finally the CONNECTION-ID. Note that the "0" means the ascii zero character (0x30), not a zero value. "1" means the ascii 1 character (0x31). "2" means the ascii 2 character (0x32).

A total of 6 keys are produced, 3 for the read side DES-EDE3 cipher and 3 for the write side DES-EDE3 function. The initialization vector is provided in the KEY-ARG-DATA. The keys that are produced are not parity adjusted. This step must be performed before proper DES keys are usable.

Recall that the MASTER-KEY is given to the server in the CLIENT-MASTER-KEY message. The CHALLENGE is given to the server by the client in the CLIENT-HELLO message. The CONNECTION-ID is given to the client by the server in the SERVER-HELLO message. This makes the resulting cipher keys a function of the original session and the current session. Note that the master key is never directly used to encrypt data, and therefore cannot be easily discovered.

The CLIENT-MASTER-KEY message must be sent after the CLIENT-HELLO message and before the CLIENT-FINISHED message. The CLIENT-MASTER-KEY

message must be sent if the SERVER- HELLO message contains a SESSION-ID-HIT value of 0.

3. CLIENT-FINISHED (Phase 2; Sent encrypted)

```
char MSG-CLIENT-FINISHED
char CONNECTION-ID[N-1]
```

The client sends this message when it is satisfied with the server. Note that the client must continue to listen for server messages until it receives a SERVER-FINISHED message. The CONNECTION-ID data is the original connection-identifier the server sent with its SERVER-HELLO message, encrypted using the agreed upon session key.

"N" is the number of bytes in the message that was sent, so "N-1" is the number of bytes in the message without the message header byte.

For version 2 of the protocol, the client must send this message after it has received the SERVER-HELLO message. If the SERVER-HELLO message SESSION-ID-HIT flag is non-zero then the CLIENT-FINISHED message is sent immediately, otherwise the CLIENT-FINISHED message is sent after the CLIENT-MASTER-KEY message.

3.5.2 Server Only Protocol Messages

There are several messages that are only generated by servers. The messages are never generated by correctly functioning clients.

1. SERVER-HELLO (Phase 1; Sent in the clear)

```
char MSG-SERVER-HELLO
char SESSION-ID-HIT
char CERTIFICATE-TYPE
char SERVER-VERSION-MSB
char SERVER-VERSION-LSB
char CERTIFICATE-LENGTH-MSB
char CERTIFICATE-LENGTH-LSB
char CIPHER-SPECS-LENGTH-MSB
char CIPHER-SPECS-LENGTH-LSB
char CONNECTION-ID-LENGTH-MSB
char CONNECTION-ID-LENGTH-LSB
char CERTIFICATE-DATA[MSB<<8|LSB]
```

```
char CIPHER-SPECS-DATA[MSB<<8|LSB]
char CONNECTION-ID-DATA[MSB<<8|LSB]
```

The server sends this message after receiving the client's CLIENT-HELLO message.

In TinySSL, the SESSION-ID-HIT value is always 0. When the SESSION-ID-HIT flag is zero, the server packages up its certificate, its cipher specs and a connection-id to send to the client. Using this information the client can generate a session key and return it to the server with the CLIENT-MASTER-KEY message.

The CONNECTION-ID-DATA is a string of randomly generated bytes used by the server and client at various points in the protocol. The CLIENT-FINISHED message contains an encrypted version of the CONNECTION-ID-DATA. The length of the CONNECTION-ID must be between 16 and than 32 bytes, inclusive.

The CIPHER-SPECS-DATA define a cipher type and key length (in bits) that the receiving end supports. Each SESSION-CIPHER-SPEC is 3 bytes long and looks like this:

```
char CIPHER-KIND-0
char CIPHER-KIND-1
char CIPHER-KIND-2
```

Where CIPHER-KIND is one of:

```
SSL_CK_RC4_128_WITH_MD5
SSL_CK_RC4_128_EXPORT40_WITH_MD5
SSL_CK_RC2_128_CBC_WITH_MD5
SSL_CK_RC2_128_CBC_EXPORT40_WITH_MD5
SSL_CK_IDEA_128_CBC_WITH_MD5
SSL_CK_DES_64_CBC_WITH_MD5
SSL_CK_DES_192_EDE3_CBC_WITH_MD5
SSL_CK_AES_128_CBC_WITH_SHA256
```

This list is not exhaustive and may be changed in the future.

TinySSL uses SSL

The SERVER-HELLO message is sent after the server receives the CLIENT-HELLO message, and before the server sends the SERVER- VERIFY message.

2. SERVER-VERIFY (Phase 1; Sent encrypted)

```
char MSG-SERVER-VERIFY
char CHALLENGE-DATA[N-1]
```

The server sends this message after a pair of session keys (SERVER-READ-KEY and SERVER-WRITE-KEY) have been agreed upon either by a session-identifier or by explicit specification with the CLIENT-MASTER-KEY message. The message contains an encrypted copy of the CHALLENGE-DATA sent by the client in the CLIENT-HELLO message.

"N" is the number of bytes in the message that was sent, so "N-1" is the number of bytes in the CHALLENGE-DATA without the message header byte.

This message is used to verify the server as follows. A legitimate server will have the private key that corresponds to the public key contained in the server certificate that was transmitted in the SERVER-HELLO message. Accordingly, the legitimate server will be able to extract and reconstruct the pair of session keys (SERVER-READ-KEY and SERVER-WRITE-KEY). Finally, only a server that has done the extraction and decryption properly can correctly encrypt the CHALLENGE-DATA. This, in essence, "proves" that the server has the private key that goes with the public key in the server's certificate.

The CHALLENGE-DATA must be the exact same length as originally sent by the client in the CLIENT-HELLO message. Its value must match exactly the value sent in the clear by the client in the CLIENT-HELLO message. The client must decrypt this message and compare the value received with the value sent, and only if the values are identical is the server to be "trusted". If the lengths do not match or the value doesn't match then the connection is to be closed by the client.

This message must be sent by the server to the client after either detecting a session-identifier hit (and replying with a SERVER-HELLO message with SESSION-ID-HIT not equal to zero) or when the server receives the CLIENT-MASTER-KEY message. This message must be sent before any Phase 2 messages or a SERVER-FINISHED message.

3. SERVER-FINISHED (Phase 2; Sent encrypted)

```
char MSG-SERVER-FINISHED
char SESSION-ID-DATA[N-1]
```

The server sends this message when it is satisfied with the clients security handshake and is ready to proceed with transmission/reception of the higher level protocols data. The `SESSION-ID-DATA` is used by the client and the server at this time to add entries to their respective session- identifier caches. The session-identifier caches must contain a copy of the `MASTER-KEY` sent in the `CLIENT-MASTER-KEY` message as the master key is used for all subsequent session key generation.

"N" is the number of bytes in the message that was sent, so "N-1" is the number of bytes in the `SESSION-ID-DATA` without the message header byte.

This message must be sent after the `SERVER-VERIFY` message.

3.5.3 Client/Server Protocol Messages

These messages are generated by both the client and the server.

1. `ERROR` (Sent clear or encrypted)

   ```
   char MSG-ERROR
   char ERROR-CODE-MSB
   char ERROR-CODE-LSB
   ```

   This message is sent when an error is detected. After the message is sent, the sending party shuts the connection down. The receiving party records the error and then shuts its connection down.

   This message is sent in the clear if an error occurs during session key negotiation. After a session key has been agreed upon, errors are sent encrypted like all other messages.

Appendix A: Protocol Constant Values

This section describes various protocol constants. A special value needs mentioning - the IANA reserved port number for "https" (HTTP using SSL). IANA has reserved port number 443 (decimal) for "https". IANA has also reserved port number 465 for "ssmtp" and port number 563 for "snntp".

Protocol Version Codes

```
#define SSL_CLIENT_VERSION 0x0002
#define SSL_SERVER_VERSION 0x0002
```

Protocol Message Codes

The following values define the message codes that are used by version 2 of the SSL Handshake Protocol.

```
#define SSL_MT_ERROR 0
#define SSL_MT_CLIENT_HELLO 1
#define SSL_MT_CLIENT_MASTER_KEY 2
#define SSL_MT_CLIENT_FINISHED 3
#define SSL_MT_SERVER_HELLO 4
#define SSL_MT_SERVER_VERIFY 5
#define SSL_MT_SERVER_FINISHED 6
#define SSL_MT_REQUEST_CERTIFICATE 7
#define SSL_MT_CLIENT_CERTIFICATE 8
```

Error Message Codes

The following values define the error codes used by the ERROR message.

```
#define SSL_PE_NO_CIPHER 0x0001
#define SSL_PE_NO_CERTIFICATE  0x0002
#define SSL_PE_BAD_CERTIFICATE 0x0004
#define SSL_PE_UNSUPPORTED_CERTIFICATE_TYPE 0x0006
```

Cipher Kind Values

The following values define the `CIPHER-KIND` codes used in the `CLIENT-HELLO` and `SERVER-HELLO` messages.

```
#define SSL_CK_RC4_128_WITH_MD5 0x01,0x00,0x80
#define SSL_CK_RC4_128_EXPORT40_WITH_MD5 0x02,0x00,0x80
#define SSL_CK_RC2_128_CBC_WITH_MD5 0x03,0x00,0x80
#define SSL_CK_RC2_128_CBC_EXPORT40_WITH_MD5 0x04,0x00,0x80
#define SSL_CK_IDEA_128_CBC_WITH_MD5 0x05,0x00,0x80
#define SSL_CK_DES_64_CBC_WITH_MD5 0x06,0x00,0x40
#define SSL_CK_DES_192_EDE3_CBC_WITH_MD5 0x07,0x00,0xC0
#define SSL_CK_AES_128_CBC_WITH_SHA256 0x11,0x00,
```

Certificate Type Codes

The following values define the certificate type codes used in the `SERVER-HELLO` and `CLIENT-CERTIFICATE` messages.

```
#define SSL_CT_X509_CERTIFICATE 0x01
```

Authentication Type Codes

The following values define the authentication type codes used in the `REQUEST-CERTIFICATE` message.

```
#define SSL_AT_MD5_WITH_RSA_ENCRYPTION 0x01
```

Upper/Lower Bounds

The following values define upper/lower bounds for various protocol parameters.

```
#define SSL_MAX_MASTER_KEY_LENGTH_IN_BITS 256
#define SSL_MAX_SESSION_ID_LENGTH_IN_BYTES 16
#define SSL_MIN_RSA_MODULUS_LENGTH_IN_BYTES 64
#define SSL_MAX_RECORD_LENGTH_2_BYTE_HEADER 32767
#define SSL_MAX_RECORD_LENGTH_3_BYTE_HEADER 16383
```

Recommendations

Because protocols have to be implemented to be of value, we recommend the following values for various operational parameters. This is only a recommendation, and not a strict requirement for conformance to the protocol.

Session-identifier Cache Timeout

Session-identifiers are kept in SSL clients and SSL servers. Session-identifiers should have a lifetime that serves their purpose (namely, reducing the number of expensive public key operations for a single client/server pairing). Consequently, we recommend a maximum session- identifier cache timeout value of 100 seconds. Given a server that can perform N private key operations per second, this reduces the server load for a particular client by a factor of 100.

4 Appendix B: Terms

Application Protocol An application protocol is a protocol that normally layers directly on top of TCP/IP. For example: HTTP, TELNET, FTP, and SMTP.

Authentication Authentication is the ability of one entity to determine the identity of another entity. Identity is defined by this document to mean the binding between a public key and a name and the implicit ownership of the corresponding private key.

Bulk Cipher This term is used to describe a cryptographic technique with certain performance properties. Bulk ciphers are used when large quantities of data are to be encrypted/decrypted in a timely manner. Examples include RC2, RC4, and IDEA.

Client In this document client refers to the application entity that initiates a connection to a server.

`CLIENT-READ-KEY` The session key that the client uses to initialize the client read cipher. This key has the same value as the `SERVER-WRITE-KEY`.

`CLIENT-WRITE-KEY` The session key that the client uses to initialize the client write cipher. This key has the same value as the `SERVER-READ-KEY`.

`MASTER-KEY` The master key that the client and server use for all session key generation. The `CLIENT-READ-KEY`, `CLIENT-WRITE-KEY`, `SERVER-READ-KEY` and `SERVER-WRITE-KEY` are generated from the `MASTER-KEY`.

`MD5` MD5 [7] is a hashing function that converts an arbitrarily long data stream into a digest of fixed size. The function has certain properties that make it useful for security, the most important of which is it's inability to be reversed. (*Note: MD5 is weaker than alternatives and is no longer recommended for new applications. We are using it here to take advantage of the existing specification.*)

Nonce A randomly generated value used to defeat "playback" attacks. One party randomly generates a nonce and sends it to the other party. The receiver encrypts it using the agreed upon secret key and returns it to the sender. Because the nonce was randomly generated by the sender this defeats playback attacks because the replayer can't know in advance the nonce the sender will generate. The receiver denies connections that do not have the correctly encrypted nonce.

Non-repudiable Information Exchange When two entities exchange information it is sometimes valuable to have a record of the communication that is non-repudiable. Neither party can then deny that the information exchange occurred. Version 2 of the SSL protocol does not support Non-repudiable information exchange.

Public Key Encryption Public key encryption is a technique that leverages asymmetric ciphers. A public key system consists of two keys: a public key and a private key. Messages encrypted with the public key can only be decrypted with the associated private key. Conversely, messages encrypted with the private key can only be decrypted with the public key. Public key encryption tends to be extremely compute intensive and so is not suitable as a bulk cipher.

Privacy Privacy is the ability of two entities to communicate without fear of eavesdropping. Privacy is often implemented by encrypting the communications stream between the two entities.

RC2, RC4 Proprietary bulk ciphers invented by RSA (There is no good reference to these as they are unpublished works; however, see [9]). RC2 is block cipher and RC4 is a stream cipher.

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 cipher A session cipher is a "bulk" cipher that is capable of encrypting or decrypting arbitrarily large amounts of data. Session ciphers are used primarily for performance reasons. The session ciphers used by this protocol are symmetric. Symmetric ciphers have the property of using a single key for encryption and decryption.

Session identifier A session identifier is a random value generated by a client that identifies itself to a particular server. The session identifier can be thought of as a handle that both parties use to access a recorded secret key (in our case a session key). If both parties remember the session identifier then the implication is that the secret key is already known and need not be negotiated.

Session key The key to the session cipher. In SSL there are four keys that are called session keys: `CLIENT-READ-KEY`, `CLIENT-WRITE-KEY`, `SERVER-READ-KEY`, and `SERVER-WRITE-KEY`.

`SERVER-READ-KEY` The session key that the server uses to initialize the server read cipher. This key has the same value as the `CLIENT-WRITE-KEY`.

`SERVER-WRITE-KEY` The session key that the server uses to initialize the server write cipher. This key has the same value as the `CLIENT-READ-KEY`.

Symmetric Cipher A symmetric cipher has the property that the same key can be used for decryption and encryption. An asymmetric cipher does not have this behavior. Some examples of symmetric ciphers: IDEA, RC2, RC4.