Traffic Analysis of an SSL/TLS Session

by Álvaro Castro-Castilla

Dec 23rd, 2014

« More entries (/)

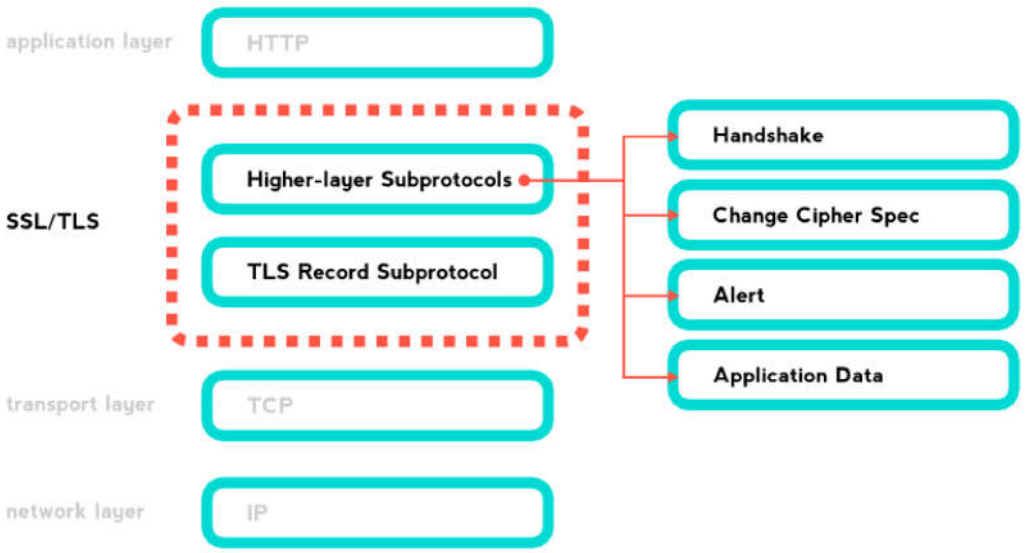
In this post I want to show what happens at the protocol level when we use SSL/TLS. For the purpose of this analysis I'll be using a non-blocking implementation of a TCP client and server based on OpenSSL (http://www.openssl.org) for the Scheme Gambit compiler (http://gambitscheme.org) that I'm currently working on.

Is important to note that SSL/TSL enables applications to be only as secure as the underlying infrastructural components (networks and hosts). SSL/TLS is a separate protocol that inserts itself between the application protocol (generally HTTP, but any other is perfectly possible) and the transport protocol (TCP). By acting as such, TLS requires very few changes to the protocols below and above, so the protocol can operate nearly transparently for users, meaning that users need not be aware of the fact that the protocol is in place. Of course this comes with some drawbacks, as it limits the protocol in some fundamental aspects (such as the inability to use UDP).

SSL/TLS has evolved considerably since its beginnings. Nowadays SSL 2.0 and 3.0 are considered insecure, so they are being replaced by the newer TLS 1.0/1.1/1.2 versions. The history of these protocols is an interesting topic.

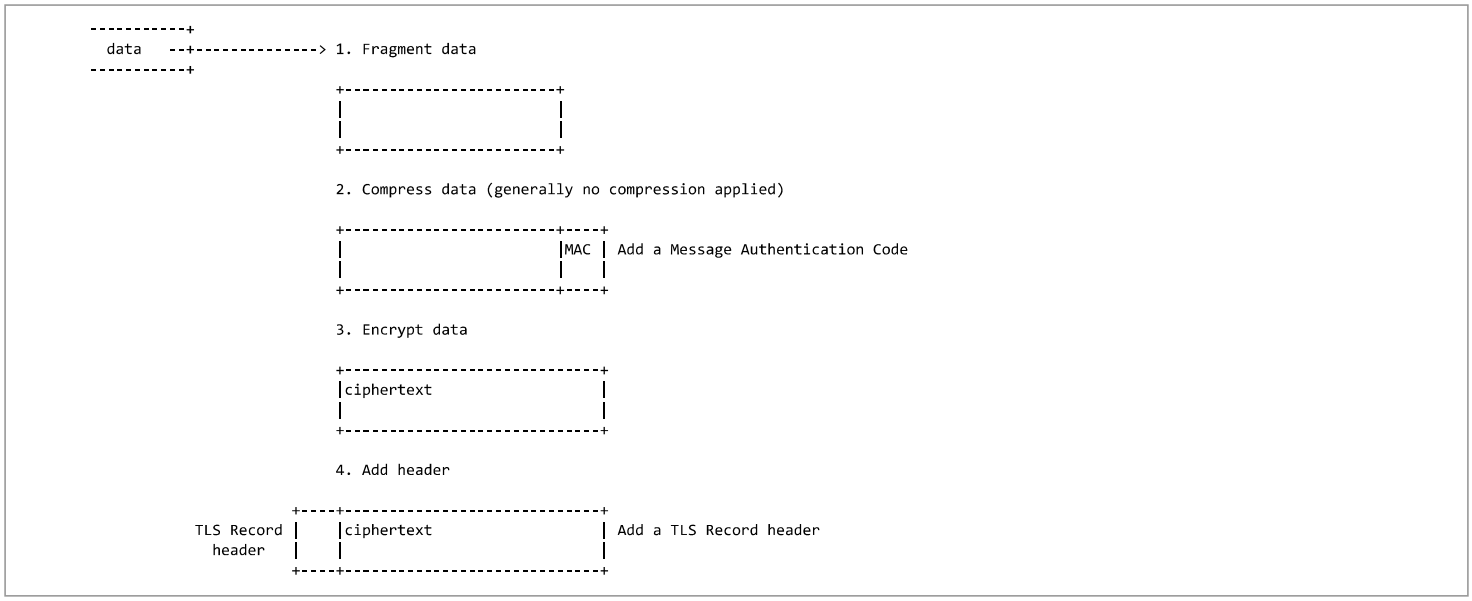
Description of the protocol

As mentioned before, the TLS protocol sits between the Application Layer and the Transport Layer. It is divided in two main sublayers. This is the general structure of the protocol, and its place in the network stack:



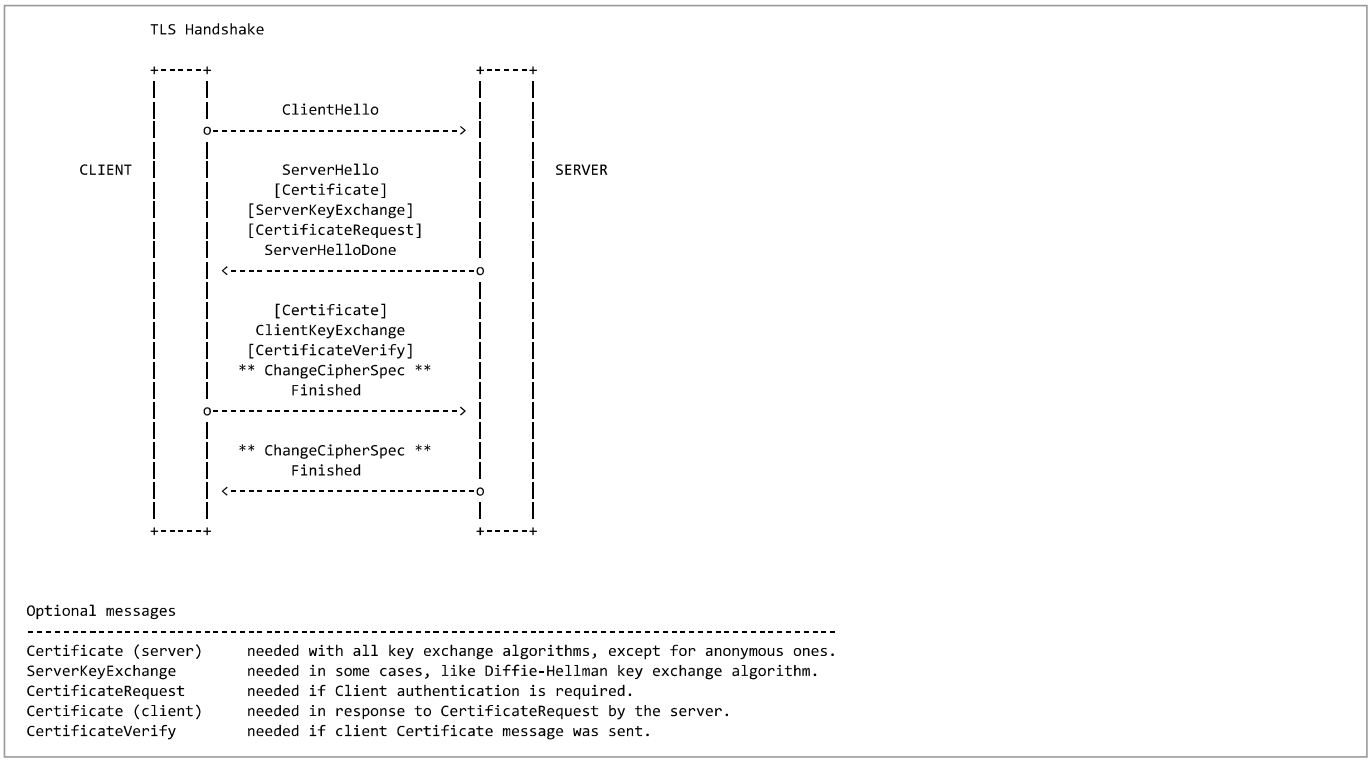
The **lower layer** is stacked on top of TCP, as it is a connection-oriented and reliable transport layer protocol. This layer consists basically of the *TLS Record Protocol*. In a nutshell, the Record Protocol first fragments the higher-layer protocol data into blocks of 2^{14} bytes or less; then optionally compresses the data, adds a Message Authentication Code and finally encrypts the data according to the cipher spec (when negotiated), adding an SSL Record header. One thing to note is that each block is packed into a structure that does not preserve client message boundaries, meaning that multiple messages of the same type may be coalesced into a single structure.

The following diagram depicts the process of building an SSL Record.



The **higher layer** is stacked on top of the SSL Record Protocol, and comprises four subprotocols. Each of these protocols has a very specific purpose, and are used at different stages of the communication:

- *Handshake Protocol*: It allows the peers to authenticate each other and to negotiate a cipher suite and other parameters of the connection. The SSL handshake protocol involves four sets of messages (sometimes called flights) that are exchanged between the client and server. Each set is typically transmitted in a separate TCP segment. The following diagram shows a summary of the process, which has several steps and offers optional ones. Please note that *ChangeCipherSpec* messages don't belong to this protocol, as they are a protocol by themselves, as seen below.



- *ChangeCipherSpec Protocol*: It makes the previously negotiated parameters effective, so communication becomes encrypted.
- *Alert Protocol*: Used for communicating exceptions and indicate potential problems that may compromise security.
- *Application Data Protocol*: It takes arbitrary data (application-layer data generally), and feeds it through the secure channel.

Several messages can be concatenated into a single Record Layer message, but those messages must belong to the same subprotocol. As a consequence of this, each of these four protocols must be self-delimiting (i.e. must include its own length field).

Record Protocol format

The TLS Record header comprises three fields, necessary to allow the higher layer to be built upon it:

- Byte 0: TLS record type

- Bytes 1-2: TLS version (major/minor)
- Bytes 3-4: Length of data in the record (excluding the header itself). The maximum supported is 16384 (16K).

```
        record type (1 byte)
       /
      /    version (1 byte major, 1 byte minor)
     /    /
    /    /        length (2 bytes)
   /    /        /
  +----+----+----+----+----+
  |    |    |    |    |    |
  |    |    |    |    |    | TLS Record header
  +----+----+----+----+----+


  Record Type Values       dec      hex
  -------------------------------------
  CHANGE_CIPHER_SPEC        20     0x14
  ALERT                     21     0x15
  HANDSHAKE                 22     0x16
  APPLICATION_DATA          23     0x17


  Version Values            dec     hex
  -------------------------------------
  SSL 3.0                   3,0  0x0300
  TLS 1.0                   3,1  0x0301
  TLS 1.1                   3,2  0x0302
  TLS 1.2                   3,3  0x0303
```

Handshake Protocol format

This is the most complex subprotocol within TLS. The specification focuses primarily on this, since it handles all the machinery necessary to establish a secure connection. The diagram below shows the general structure of Handshake Protocol messages. There are 10 handshake message types in the TLS specification (not counting extensions), so the specific format of each one will be described below.

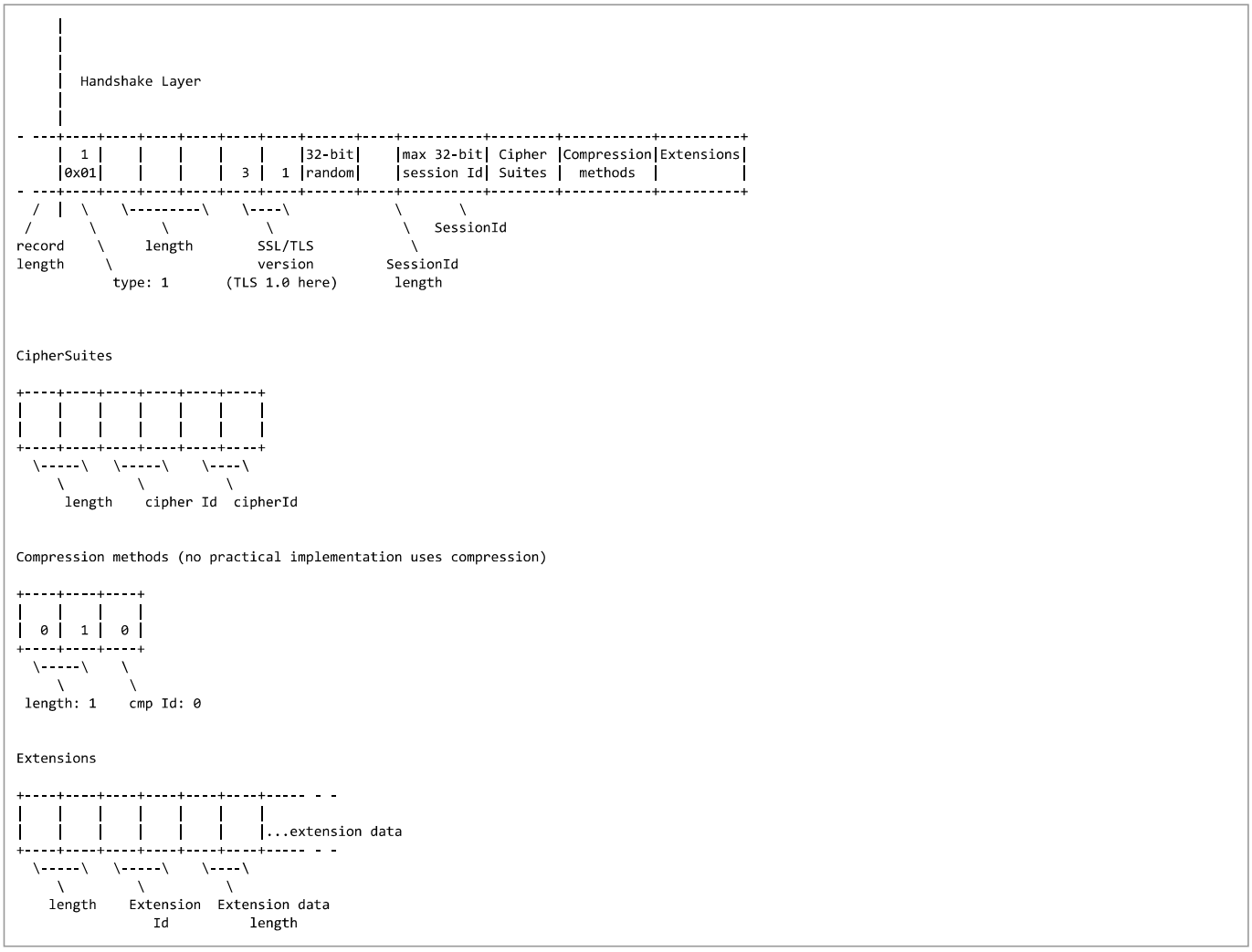
```
                        |
                        |
                        |
       Record Layer     |  Handshake Layer
                        |                                  |
                        |                                  |  ...more messages
+----+----+----+----+----+----+----+----+----+------ - - - -+--
| 22 |    |    |    |    |    |    |    |    |              |
|0x16|    |    |    |    |    |    |    |    |message       |
+----+----+----+----+----+----+----+----+----+------ - - - -+--
  /              /      | \    \----\-----\                |
 /              /       |  \         \
type: 22       /        |   \         handshake message length
              /             type
             /
         length: arbitrary (up to 16k)


 Handshake Type Values    dec      hex
 -------------------------------------
 HELLO_REQUEST              0     0x00
 CLIENT_HELLO               1     0x01
 SERVER_HELLO               2     0x02
 CERTIFICATE               11     0x0b
 SERVER_KEY_EXCHANGE       12     0x0c
 CERTIFICATE_REQUEST       13     0x0d
 SERVER_DONE               14     0x0e
 CERTIFICATE_VERIFY        15     0x0f
 CLIENT_KEY_EXCHANGE       16     0x10
 FINISHED                  20     0x14
```

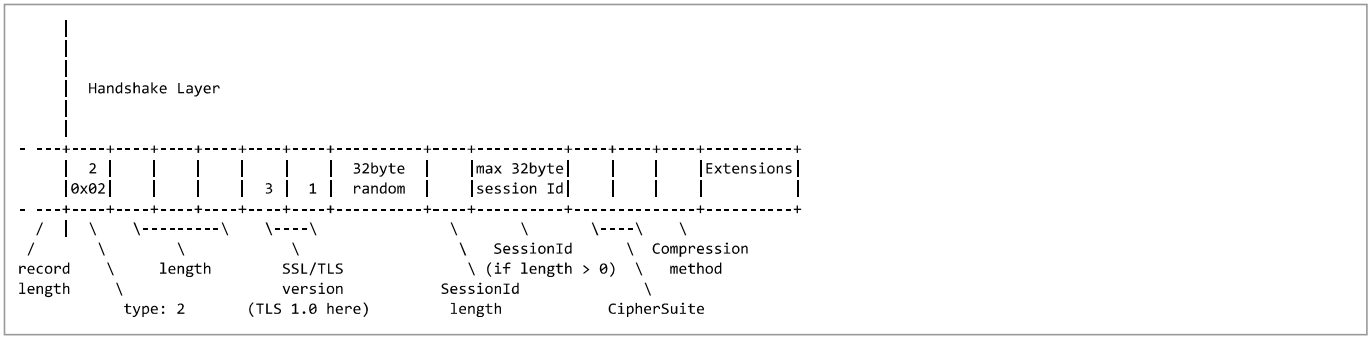
- **HelloRequest**: Allows a server to restart the handshake negotiation. Not used very often. If a connection has been up for long enough time that its security is weakened (in the order of hours), the server can use this message to force a client to renegotiate new session keys.

```
     |
     |
     |
     |  Handshake Layer
     |
     |
- ---+----+----+----+----+
     |    |    |    |    |
   4 |  0 |  0 |  0 |  0 |
- ---+----+----+----+----+
  /  |  \    \---------\
 /       \        \
record    \    length: 0
length     \
            type: 0
```

- **ClientHello**: This message typically begins a TLS handshake negotiation. It is sent with a list of client-supported cipher suites, for the server to pick the best suiting one (preferably the strongest), a list of compression methods, and a list of extensions. It gives also the possibility to the client of restarting a previous session, through the inclusion of a *SessionId* field.



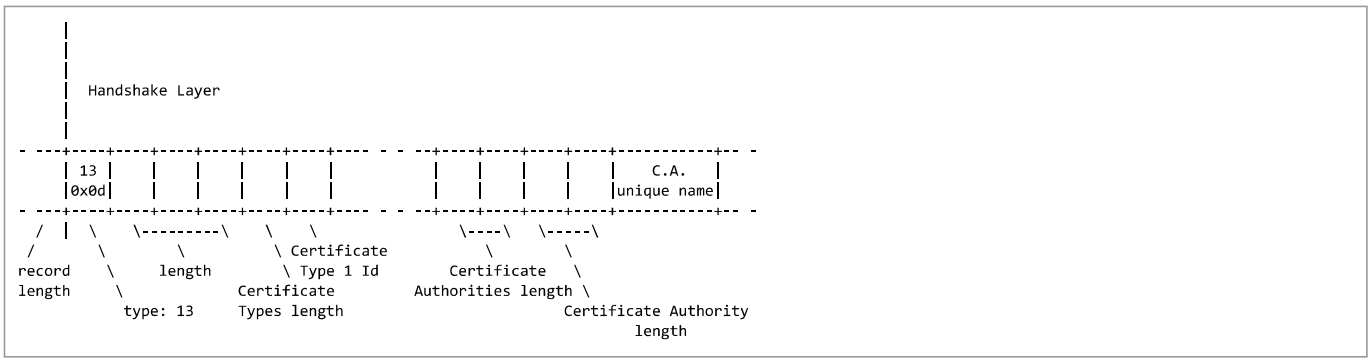
- **ServerHello**: The ServerHello message is very similar to the ClientHello message, with the exception that it only includes one CipherSuite and one Compression method. If it includes a SessionId (i.e. SessionId Length is > 0), it signals the client to attempt to reuse it in the future.



- **Certificate**: The body of this message contains a chain of public key certificates. Certificate chains allows TLS to support certificate hierarchies and PKIs (Public Key Infrastructures).

- **ServerKeyExchange**: This message carries the keys exchange algorithm parameters that the client needs from the server in order to get the symmetric encryption working thereafter. It is optional, since not all key exchanges require the server explicitly sending this message. Actually, in most cases, the Certificate message is enough for the client to securely communicate a premaster key with the server. The format of those parameters depends exclusively on the selected CipherSuite, which has been previously set by the server via the ServerHello message.

- **CertificateRequest**: It is used when the server requires client identity authentication. Not commonly used in web servers, but very important in some cases. The message not only asks the client for the certificate, it also tells which certificate types are acceptable. In addition, it also indicates which Certificate Authorities are considered trustworthy.



- **ServerHelloDone**: This message finishes the server part of the handshake negotiation. It does not carry any additional information.

- **ClientKeyExchange**: It provides the server with the necessary data to generate the keys for the symmetric encryption. The message format is very similar to ServerKeyExchange, since it depends mostly on the key exchange algorithm picked by the server.

- **CertificateVerify**: This message is used by the client to prove the server that it possesses the private key corresponding to its public key certificate. The message holds hashed information digitally signed by the client. It is required if the server issued a CertificateRequest to the client, so that it had to send a Certificate that needs to be verified. Once again, the exact size and structure of the information depends on the agreed algorithm. In all cases, the information that serves as input to the hash functions is the same.

- **Finished**: This message signals that the TLS negotiation is complete and the CipherSuite is activated. It should be sent already encrypted, since the negotiation is successfully done, so a ChangeCipherSpec protocol message must be sent before this one to activate the encryption. The Finished message contains a hash of all previous handshake messages combined, followed by a special number identifying server/client role, the master secret and padding. The resulting hash is different from the CertificateVerify hash, since there have been more handshake messages.

ChangeCipherSpec Protocol format

This is the simplest protocol: it has only one message. The reason why this message must be a separate protocol instead of being part of the Handshake Protocol is because of the Record Layer encapsulation. The TLS protocol applies encryption to entire Record Layer messages at once. The ChangeCipherSpec message signals the activation of encryption, and since encryption cannot be applied to parts of a message it is impossible for any other message to follow a ChangeCipherSpec one. The best way to avoid those combinations is by elevating this message to the protocol status.

This is how the only ChangeCipherSpec message is structured:

Alert Protocol format

The Alert Protocol is also rather simple. It defines two fields: severity level and alert description. The first field indicates the severity of the alert (1 for warning, 2 for fatal), while the second field encodes the exact condition. The supported alert descriptions depend on the SSL/TLS version.

```
                        |
                        |
                        |
       Record Layer     |  Alert Layer
                        |
                        |
+----+----+----+----+----+----+----+
| 21 |    |    |    |    |    |    |
|0x15|    |    |  0 |  2 |    |    |
+----+----+----+----+----+----+----+
  /              /      |
 /              /       |
type: 21       /        |
              /
             /
         length: 2


Alert severity               dec     hex
-----------------------------------------
WARNING                        1    0x01
FATAL                          2    0x02


TLS 1.0 Alert descriptions   dec     hex
-----------------------------------------

CLOSE_NOTIFY                   0    0x00
UNEXPECTED_MESSAGE            10    0x0A
BAD_RECORD_MAC                20    0x14
DECRYPTION_FAILED             21    0x15
RECORD_OVERFLOW               22    0x16
DECOMPRESSION_FAILURE         30    0x1E
HANDSHAKE_FAILURE             40    0x28
NO_CERTIFICATE                41    0x29
BAD_CERTIFICATE               42    0x2A
UNSUPPORTED_CERTIFICATE       43    0x2B
CERTIFICATE_REVOKED           44    0x2C
CERTIFICATE_EXPIRED           45    0x2D
CERTIFICATE_UNKNOWN           46    0x2E
ILLEGAL_PARAMETER             47    0x2F
UNKNOWN_CA                    48    0x30
ACCESS_DENIED                 49    0x31
DECODE_ERROR                  50    0x32
DECRYPT_ERROR                 51    0x33
EXPORT_RESTRICTION            60    0x3C
PROTOCOL_VERSION              70    0x46
INSUFFICIENT_SECURITY         71    0x47
INTERNAL_ERROR                80    0x50
USER_CANCELLED                90    0x5A
NO_RENEGOTIATION             100    0x64
```

ApplicationData Protocol format

The mission of this protocol is to properly encapsulate the data coming from the Application Layer of the network stack, so it can seamlessly be handled by the underlying protocol (TCP) without forcing changes in any of those layers. The format of the messages in this protocols follows the same structure as the previous ones.

```
                        |
                        |
                        |
       Record Layer     |  ApplicationData Layer (encrypted)
                        |
                        |
+----+----+----+----+----+----+----+--- - - - - - - --+---------+
| 23 |    |    |    |       length-delimited data     |         |
|0x17|    |    |    |    |    |    |                  |   MAC   |
+----+----+----+----+----+----+----+--- - - - - - - --+---------+
  /              /      |
 /              /       |
type: 23       /        |
              /
             /
         length: arbitrary (up to 16k)
```

Analyzing SSL/TLS traffic

The main purpose of the traffic analysis will be testing the TLS client implementation I'm currently working on for the Gambit Scheme compiler (http://gambitscheme.org). I will be using Wireshark (http://wireshark.org) for the packet capture, the client will be Scheme-based with the work-in-progress support for SSL/TLS, and the test server (with a certificate) that is part of the OpenSSL distribution. The server is listening on port 443, and all communication will be done through the loopback device. The easiest way to restrict your view to TLS packets in Wireshark is to use the protocol filter 'ssl'.

First flight (client –> server)

Once the server is running and waiting for connection, the client can initiate it. This is the first packet, sent by the client:

```
Full contents of the packet

0000   02 00 00 00 45 00 00 98 13 ed 40 00 40 06 00 00  ....E.....@.@...
0010   7f 00 00 01 7f 00 00 01 ec 26 01 bb 43 7c ee 74  .........&..C|.t
0020   60 b5 50 0a 80 18 31 d7 fe 8c 00 00 01 01 08 0a  `.P...1.........
0030   21 62 1e 1e 21 62 1e 1e 16 03 01 00 5f 01 00 00  !b..!b......_...
0040   5b 03 01 54 9a ab 72 98 65 11 2f da 9e cf c9 db  [..T..r.e./.....
0050   6c bd 4b 4c 56 4b 0c a5 68 2b aa 60 1f 38 66 e7  l.KLVK..h+.`.8f.
0060   87 46 b2 00 00 2e 00 39 00 38 00 35 00 16 00 13  .F.....9.8.5....
0070   00 0a 00 33 00 32 00 2f 00 9a 00 99 00 96 00 05  ...3.2./........
0080   00 04 00 15 00 12 00 09 00 14 00 11 00 08 00 06  ................
0090   00 03 00 ff 01 00 00 04 00 23 00 00              .........#..

------------

Family: IP

0000   02 00 00 00                                      ....

IP protocol

0000   45 00 00 98 13 ed 40 00 40 06 00 00 7f 00 00 01  E.....@.@.......
0010   7f 00 00 01                                      ....

TCP protocol

0000   ec 26 01 bb 43 7c ee 74 60 b5 50 0a 80 18 31 d7  .&..C|.t`.P...1.
0010   fe 8c 00 00 01 01 08 0a 21 62 1e 1e 21 62 1e 1e  ........!b..!b..

------------

TLSv1 protocol

0000   16 03 01 00 5f 01 00 00 5b 03 01 54 9a ab 72 98  ...._...[..T..r.
0010   65 11 2f da 9e cf c9 db 6c bd 4b 4c 56 4b 0c a5  e./.....l.KLVK..
0020   68 2b aa 60 1f 38 66 e7 87 46 b2 00 00 2e 00 39  h+.`.8f..F.....9
0030   00 38 00 35 00 16 00 13 00 0a 00 33 00 32 00 2f  .8.5.......3.2./
0040   00 9a 00 99 00 96 00 05 00 04 00 15 00 12 00 09  ................
0050   00 14 00 11 00 08 00 06 00 03 00 ff 01 00 00 04  ................
0060   00 23 00 00                                      .#..

TLSv1 Record protocol

0000   16 03 01 00 5f                                   ...._

       16             Handshake protocol type
       03 01          SSL version (TLS 1.0)
       5f             Record length (95 bytes)

TLSv1 Handshake protocol

0000   01 00 00 5b 03 01 54 9a ab 72 98 65 11 2f da 9e  ...[..T..r.e./..
0010   cf c9 db 6c bd 4b 4c 56 4b 0c a5 68 2b aa 60 1f  ...l.KLVK..h+.`.
0020   38 66 e7 87 46 b2 00 00 2e 00 39 00 38 00 35 00  8f..F.....9.8.5.
0030   16 00 13 00 0a 00 33 00 32 00 2f 00 9a 00 99 00  ......3.2./.....
0040   96 00 05 00 04 00 15 00 12 00 09 00 14 00 11 00  ................
0050   08 00 06 00 03 00 ff 01 00 00 04 00 23 00 00     ............#..

       01             ClientHello message type
       00 00 5b       Message length
       03 01          SSL version (TLS 1.0)
       54 .. b2       32-bytes random number
       00             Session Id length
       00 2e          Cipher Suites length (46 bytes, 23 suites)
       00 39 .. ff    23 2-byte Cipher Suite Id numbers
       01             Compression methods length (1 byte)
       00             Compression method (null)
       00 04          Extensions length (4 bytes)
       00 23          SessionTicket TLS extension Id
       00 00          Extension data length (0)
```

Second flight (server –> client)

The first flight comprised only one TLS handshake message (ClientHello), which is sent from the client to the server. However, the next TCP packet, sent by the server to the client in response to ClientHello carries 3 Handshake messages. These messages are ServerHello, Certificate and ServerHelloDone (no ServerKeyExchange or CertificateRequest are sent). In the next packet and thereafter, I will omit the lower stack protocols (TCP/IP).

```
ServerHello message

0000   16 03 01 00 35 02 00 00 31 03 01 54 9a ab 72 85  ....5...1..T..r.
0010   91 a4 a7 a9 27 fe 3d e4 da f6 38 a5 aa 6e 5a 2f  ....'.=...8..nZ/
0020   31 90 5b 41 b0 5d de d8 9d ae f6 00 00 35 00 00  1.[A.].......5..
0030   09 ff 01 00 01 00 00 23 00 00                    .......#..

       16             Handshake protocol type
       03 01          SSL version (TLS 1.0)
       35             Record length (53 bytes)

       02             ServerHello message type
       00 00 31       Message length (49 bytes)
       03 01          SSL version (TLS 1.0)
       54 9a ab 72    First 4 bytes of random (Unix time)
       85 .. f6       Last 28 bytes of the random number
       00             Session Id length
       00 35          Selected Cipher Suite (RSA with AES-256-CBC SHA)
       00             Selected compression method (null)
       00 09          Extensions length
       ff 01 00 01 00 Extension (Renegotiation Info)
       00 23 00 00    Extension (SessionTicket TLS)


Certificate message

0000   16 03 01 01 e4 0b 00 01 e0 00 01 dd 00 01 da 30  ...............0
0010   82 01 d6 30 82 01 3f 02 01 01 30 0d 06 09 2a 86  ...0..?...0...*.
0020   48 86 f7 0d 01 01 04 05 00 30 45 31 0b 30 09 06  H........0E1.0..
0030   03 55 04 06 13 02 41 55 31 13 30 11 06 03 55 04  .U....AU1.0...U.
0040   08 13 0a 53 6f 6d 65 2d 53 74 61 74 65 31 21 30  ...Some-State1!0
0050   1f 06 03 55 04 0a 13 18 49 6e 74 65 72 6e 65 74  ...U....Internet
0060   20 57 69 64 67 69 74 73 20 50 74 79 20 4c 74 64   Widgits Pty Ltd
0070   30 1e 17 0d 39 39 30 35 30 31 30 31 32 36 33 35  0...990501012635
0080   5a 17 0d 39 39 30 35 33 31 30 31 32 36 33 35 5a  Z..990531012635Z
0090   30 22 31 0b 30 09 06 03 55 04 06 13 02 44 45 31  0"1.0...U....DE1
00a0   13 30 11 06 03 55 04 03 13 0a 54 65 73 74 73 65  .0...U....Testse
00b0   72 76 65 72 30 81 9f 30 0d 06 09 2a 86 48 86 f7  rver0..0...*.H..
00c0   0d 01 01 01 05 00 03 81 8d 00 30 81 89 02 81 81  ..........0.....
00d0   00 fa 23 7a 03 2a 27 b1 c3 09 64 ce 36 ab eb d0  ..#z.*'...d.6...
00e0   08 16 75 54 68 6f 39 2e d0 9e 81 ed 91 f8 2b 48  ..uTho9.......+H
00f0   0e 59 10 63 0e bc ff c3 1b 4f 7a 2e d2 97 45 01  .Y.c.....Oz...E.
0100   c2 fd 20 68 98 63 76 34 48 73 3d 3e a1 74 d1 13  .. h.cv4Hs=>.t..
0110   b5 30 2b 4d a6 a4 e7 17 74 9c 2e 96 e6 82 01 a3  .0+M....t.......
0120   2a 29 66 59 89 f6 6a 2e de 99 d8 cc 8d 75 4b b7  *)fY..j......uK.
0130   35 96 db 11 a0 20 60 13 59 03 77 d8 a8 1f 26 78  5.... `.Y.w...&x
0140   38 8d 78 b5 52 31 22 c8 b8 64 c3 46 5f d4 8f e0  8.x.R1"..d.F_...
0150   83 02 03 01 00 01 30 0d 06 09 2a 86 48 86 f7 0d  ......0...*.H...
0160   01 01 04 05 00 03 81 81 00 c8 0c fa c6 c0 93 c0  ................
0170   df 8d 27 da f9 17 f6 81 c1 97 99 ba ef 64 0c ca  ..'..........d..
0180   cc 2f b9 45 4d e4 6a af cd cb 12 17 00 67 28 f5  ./.EM.j......g(.
0190   d6 63 a3 3c d6 7c df f1 b8 6b a9 e5 ba 05 93 e2  .c.<.|...k......
01a0   ab 3f ec 5d 82 c6 aa 18 7b 32 ce 58 04 a2 ac f8  .?.]....{2.X....
01b0   7a 4a 8b 8d 07 95 6e 7a 23 df 7f 61 54 55 3d 32  zJ....nz#..aTU=2
01c0   13 e2 e8 95 0b 3f 18 d7 2a e9 a3 7d 7d 8b 2c d9  .....?..*..}}.,.
01d0   22 91 6e 69 bb 3f 03 7f 75 22 5f 41 22 68 9b dd  ".ni.?..u"_A"h..
01e0   ec 4c 0f f0 9e f9 b6 25 13                       .L.....%.

       16             Handshake protocol type
       03 01          SSL version (TLS 1.0)
       01 e4          Record length (443 bytes)

       0b             Certificate message type
       00 01 e0       Message length
       00 01 dd       Certificates length
       00 .. 13       Certificates data


ServerHelloDone message

0000   16 03 01 00 04 0e 00 00 00                       .........

       16             Handshake protocol type
       03 01          SSL version (TLS 1.0)
       00 04          Record length (4 bytes)

       0e             ServerHelloDone message
       00 00 00       Message length
```

Third flight (client –> server)

This all seems right and is consistent with the protocol outlined above. The client and the server has now agreed on the algorithms to use (RSA for key exchange, AES-256-CBC for symmetric encryption and SHA for message hashing), the compression (no compression) and the TLS extensions in use (SessionTicket TLS, Renegotiation Info). Also, the client is now in possession of the server's certificates, so it can decide whether it will trust the server or not. The next packet is sent by the client and carries the following messages: ClientKeyExchange, ChangeCipherSpec, Finished (which is already encrypted).

```
ClientKeyExchange message

0000   16 03 01 00 86 10 00 00 82 00 80 2d 28 e4 30 eb  ...........-(.0.
0010   31 35 b0 4b 5e 4c 4d c6 ee 01 f5 33 e7 f8 3f 9b  15.K^LM....3..?.
0020   d7 53 fc 5c e0 2d d6 12 ba 55 f8 46 ab 73 d8 3d  .S.\.-...U.F.s.=
0030   b0 0a f7 03 7f 58 e0 32 8f 91 1f b8 cf 56 aa 89  .....X.2.....V..
0040   9e 27 84 08 ec 78 f8 74 0c d3 80 f2 ec 04 65 e1  .'...x.t......e.
0050   3e 92 91 52 b5 5e aa 67 e9 e6 40 e9 10 67 3c 3f  >..R.^.g..@..g<?
0060   73 f7 62 4a 0c 42 30 c1 06 6f 53 2f c2 6b d5 c8  s.bJ.B0..oS/.k..
0070   67 6f 06 d7 92 86 6e 1d 4d dd 6b 3f b0 26 6c 25  go....n.M.k?.&l%
0080   2c d8 81 5a 80 e0 e2 cc d1 62 9c                 ,..Z.....b.

       16             Handshake protocol type
       03 01          SSL version (TLS 1.0)
       00 86          Record length (134 bytes)

       10             ClientKeyExchange message type
       00 00 82       Message length (130 bytes)
       00 .. 9c       RSA encrypted key data (premaster secret)


ChangeCipherSpec message

0000   14 03 01 00 01 01                                ......

       14             ChangeCipherSpec protocol type
       03 01          SSL version (TLS 1.0)
       00 01          Message length (1 byte)

       01             ChangeCipherSpec message


Finished message (encrypted)

0000   16 03 01 00 30 0c c1 86 73 a4 a3 26 62 30 21 7f  ....0...s..&b0!.
0010   c3 2f 1a 83 34 2d 57 f0 e2 0d 37 d4 51 66 08 22  ./..4-W...7.Qf."
0020   b0 ea b4 a4 1e 81 2a fd 5f 07 47 9f b7 2c 0a dc  ......*._.G..,..
0030   65 08 77 40 2a                                   e.w@*

       16             Handshake protocol type
       03 01          SSL version (TLS 1.0)
       00 30          Message length (48 bytes)

       0c .. 2a       Encrypted Finished message
```

Fourth flight (server –> client)

After the client sends the ChangeCipherSpec and Finished messages, the server is expected to do the same, in order to bilaterally start encrypted communication with the secret symmetric keys and all the agreed Cipher Suite parameters. The server must send its own ChangeCipherSpec and Finished messages so the handshake process can be considered successful. One very interesting thing that happens in this message is that we see one of the extensions in action. The extension is called *Transport Layer Security (TLS) Session Resumption without Server-Side State*, which states clearly what it does. You should recall that it was requested as part of our ClientHello, and fulfilled by the server in its ServerHello. For information about messages related to this extension, we need to look for the RFC5077 (http://www.ietf.org/rfc/rfc5077.txt) specification. As described in the document, this extension handshake message has been assigned the number 4.

```
NewSessionTicket message

0000   16 03 01 00 aa 04 00 00 a6 00 00 00 00 00 a0 f7  ................
0010   2f 0c fd be ce f7 96 86 ca fd da 58 d6 16 b3 3c  /..........X...<
0020   89 1a a5 a2 af 3c 80 50 7b 99 71 05 3b 0e d3 27  .....<.P{.q.;..'
0030   75 78 0d 0a 20 6c e7 1c ce 7b 5d 52 ad f1 04 88  ux.. l...{]R....
0040   ec fa 04 c9 6a 74 fc 7b 3d 99 aa 8a ec 7a a3 18  ....jt.{=....z..
0050   81 63 2f db b0 16 5b 49 63 f4 53 bc 57 18 27 37  .c/...[Ic.S.W.'7
0060   f2 7f 66 e6 4d 46 59 2d 17 39 d5 79 a4 49 4d 93  ..f.MFY-.9.y.IM.
0070   d2 80 34 8b 49 f5 31 72 7f 7b 41 46 37 9b ae a9  ..4.I.1r.{AF7...
0080   3c f0 6f 2e 7f 75 e3 bf 2f d8 fc a4 be cb 2c 84  <.o..u../.....,.
0090   01 b2 25 01 23 91 6e c0 c1 09 9d 42 c8 b8 e6 1b  ..%.#.n....B....
00a0   fe 1e ed b3 52 7f 25 90 ae fc 34 f5 96 1b f0     ....R.%...4....

       16             Handshake protocol type
       03 01          SSL version (TLS 1.0)
       aa 04          Message length (170 bytes)

       04             New Session Ticket message type (extension)
       00 00 a6       Message length (166 bytes)
       00 .. f0       Session Ticket data


ChangeCipherSpec message

0000   14 03 01 00 01 01                                ......

       14             ChangeCipherSpec protocol type
       03 01          SSL version (TLS 1.0)
       00 01          Message length

       01             ChangeCipherSpec message


Finished message (encrypted)

0000   16 03 01 00 30 6d 09 0e 9f dd 09 03 2f 84 65 f8  ....0m....../.e.
0010   94 0f d6 7b 4b 54 31 a1 25 a4 27 03 ae c3 4e af  ...{KT1.%.'...N.
0020   27 04 32 5a 1f 29 90 fa 0a 4b 89 2f af d8 88 99  '.2Z.)...K./....
0030   41 de dd 89 3f                                   A...?

       16             Handshake protocol type
       03 01          SSL version (TLS 1.0)
       00 30          Message length (48 bytes)

       6d .. 3f       Encrypted Finished message
```

Application data (client <–> server)

At this point, the client and server are done with the handshake. Encrypted communication is on place, and application data can be transmitted securely. This is an example record, of which several can be carried in one single TCP packet:

```
ApplicationData message

0000   17 03 01 00 30 5d 15 3d a2 40 ef d2 01 25 ca 54  ....0].=.@...%.T
0010   26 5f 5d b0 d2 2f 2f 6d 2d ec 56 85 b0 4c a9 bf  &_].././/m-.V..L..
0020   eb 97 be 31 ad cd de 3a b4 71 1e c8 53 96 0b 2d  ...1...:.q..S..-
0030   c3 91 3d a2 15                                   ..=..

       17             ApplicationData protocol type
       03 01          SSL version (TLS 1.0)
       00 30          Message length (48 bytes)

       5d .. 15       Encrypted application data
```

Closing connection

Since we are already in an encrypted connection, the only way to really know what is being sent within packets is to make Wireshark or similar tools aware of the keys used in the transmission. Even though this is possible (http://packetpushers.net/using-wireshark-to-decode-ssltls-packets/), I think for the purpose of this analysis it is enough to know that the client sends an alert message when the connection is asked to be closed actively by the client or server. The type of this Alert message should be CloseNotify (type 0), but we won't be able to see it from the raw data. In this case, the client is the sender of the following Alert message:

```
Alert message

0000   15 03 01 00 20 3e 2e 43 30 49 49 6a f6 37 69 eb  .... >.C0IIj.7i.
0010   0d dd c3 e2 d3 e1 5d e3 4e b3 e2 22 9d 85 f9 c4  ......].N.."....
0020   59 b3 41 a9 86                                   Y.A..

       15             Alert protocol type
       03 01          SSL version (TLS 1.0)
       00 20          Message length (32 bytes)

       3e .. 86       Encrypted alert message
```

Conclusion

As far as I can tell from the traffic analysis, the library and the client implementation is doing as expected from the TLS 1.0 specification. I hope this was useful for you to gain some insight of the inner workings of SSL/TLS protocol. I find inspecting hex raw data (specially with such powerful tools as Wireshark) a very instructive and fun way of working with networks.

Did you find it useful? Please share!

Tweet 85
(http://twitter.com/share)

Related Posts

- Building a Distributed Fault-Tolerant Key-Value Store (/2015/04/12/building-a-distributed-fault-tolerant-key-value-store/)
- Lightweight object systems for Scheme (/2014/05/22/lightweight-object-systems-for-scheme/)
- Essential CSS positioning (/2013/11/27/essential-css-positioning/)
- The Path of the Command Line (/2013/10/18/the-path-of-the-command-line/)

network-programming (/blog/categories/network-programming/) **reference** (/blog/categories/reference/) **security** (/blog/categories/security/)

« SchemeSpheres Experimental: R7RS libraries and more Scheme implementations (/2014/10/24/schemespheres-experimental-r7rs-libraries-and-more-scheme-implementations/) Building a Distributed Fault-Tolerant Key-Value Store » (/2015/04/12/building-a-distributed-fault-tolerant-key-value-store/)

Comments