Appendix C

TLS 1.2 Protocol Execution Transcript

In Section 2.3, we overviewed a relatively simple protocol execution transcript for SSL 3.0. In this appendix, we do something similar for TLS 1.2. Since TLS 1.2 is considerably more involved than SSL 3.0, we use a rather long appendix for this purpose. The starting point is the same as in Section 2.3: We use a standard client (Firefox) to connect to a standard Web server (Apache) running at `www.esecurity.ch`, and we use Wireshark to log the transcript of the TLS 1.2 handshake protocol execution and respective session establishment. We only focus on TLS-related data, so all lower-layer protocol data units (in particular, the ones that refer to TCP, IP, and Ethernet) are ignored and not discussed here. The message flows are illustrated in Figures 2.5 and 3.6. Also, we don't repeat the explanations made in Chapters 2 and 3 here. Instead, we keep the explanations as short and as possible, and we refer to these explanations and RFC documents where appropriate.

In our example, the transcript starts with a CLIENTHELLO message that is sent from the client to the server. This message is transmitted in a TLS record of its own. Using hexadecimal notation, this record looks as follows:

```
16 03 01 00 be 01 00 00   ba 03 03 87 23 8f 22 bb
31 bc 06 c2 ae b7 5e b2   3f 49 68 6e e8 f3 a6 af
b4 72 ec 01 4c cd 4d 6c   06 51 13 00 00 1e c0 2b
c0 2f cc a9 cc a8 c0 2c   c0 30 c0 0a c0 09 c0 13
c0 14 00 33 00 39 00 2f   00 35 00 0a 01 00 00 73
00 00 00 15 00 13 00 00   10 77 77 77 2e 65 73 65
63 75 72 69 74 79 2e 63   68 00 17 00 00 ff 01 00
01 00 00 0a 00 08 00 06   00 17 00 18 00 19 00 0b
00 02 01 00 00 23 00 00   33 74 00 00 00 10 00 0e
00 0c 02 68 32 08 68 74   74 70 2f 31 2e 31 00 05
00 05 01 00 00 00 00 00   0d 00 18 00 16 04 01 05
```

DRAFT VERSION

```
01 06 01 02 01 04 03 05   03 06 03 02 03 05 02 04
02 02 02
```

The record starts with a type field that holds the value 0x16 (representing 22 in decimal notation), and hence stands for a TLS record that may comprise one or several handshake messages (in this case, the record only comprises a single message). The next two bytes hold 0x0301 and refer to the TLS version 1.0. The last two bytes of the TLS record header hold 0x00be, referring to the byte length of the fragment that comprises the CLIENTHELLO message. So the message is 190 bytes long, and hence the subsequent 190 bytes represent it. It starts with 0x01 standing for the handshake message type 1 (referring to a CLIENTHELLO message), 0x0000ba standing for a the length of the message (i.e., 186 bytes), and 0x0303 standing for TLS 1.2. So in contrast to the record header, the client signals in the CLIENTHELLO message that it supports TLS 1.2, and hence that it can use TLS 1.2 to establish a session and to exchange data accordingly. The following 32 bytes—ranging from 0x8723 to 0x5113—represent the random value chosen by the client (remember that the first 4 bytes represent timing information and that only the subsequent 28 bytes represent the actual random value). Because there is no TLS session to resume, the session ID length that follows is zero (i.e., 0x00) and hence no session ID is appended.

Table C.1
15 Cipher Suites Supported by the Client

Code	Cipher Suite
0xc02b	TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256
0xc02f	TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256
0xcca9	TLS_ECDHE_ECDSA_WITH_CHACHA20_POLY1305_SHA256 *
0xcca8	TLS_ECDHE_RSA_WITH_CHACHA20_POLY1305_SHA256 *
0xc02c	TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384
0xc030	TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384
0xc00a	TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA
0xc009	TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA
0xc013	TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA
0xc014	TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA
0x0033	TLS_DHE_RSA_WITH_AES_128_CBC_SHA
0x0039	TLS_DHE_RSA_WITH_AES_256_CBC_SHA
0x002f	TLS_RSA_WITH_AES_128_CBC_SHA
0x0035	TLS_RSA_WITH_AES_256_CBC_SHA
0x000a	TLS_RSA_WITH_3DES_EDE_CBC_SHA

The next next 2 bytes, i.e., 0x001e, (30 in decimal notation) indicate that the subsequent 30 bytes refer to the 15 cipher suites that are supported by the

DRAFT VERSION

client. They are summarized in Table C.1. In the first column of this table, the code (represented by a pair of bytes) refers to a particular cipher suite outlined in Appendix A. The two cipher suites that use ChaCha20 and Poly1305 are introduced in RFC 7905 (they are marked with a star in Table C.1). Because TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256 is the cipher suite preferred by the client (as it appears first in the list) that is also supported by the server, this is the cipher suite that is going to be selected afterwards.

After the list of cipher suites, the byte 0x01 refers to one compression method that is supported by the client, and the byte 0x00 refers to this compression method which is null (standing for no compression). All bytes that follow refer to TLS extensions. The length of the extensions is indicated with the first two bytes, i.e., 0x0073 (115 in decimal notation). So the next 115 bytes refer to a total of 10 extensions that are included in the CLIENTHELLO message. You may refer to Table 3.11 and the subsequent explanations to get some further information about the extensions. We briefly outline them in the order in which they appear in the transcript (but, in principle, the order is arbitrary and does not matter).

The first extension, i.e., `server_name`, refers to the SNI extension and comprises 25 bytes that are summarized in Table C.2. In the table header, # refers to the number of bytes that are occupied by this field. In the description column, a number in brackets refers to the decimal value of the respective bytes (e.g., 0x0015 refers to 21). If the value is standing for a type, then the respective type name is usually added, as well. With these informations, it should be possible to read and understand the byte encoding of the `server_name` extension. In this particular case, the extension is used to tell the server that the client wants to establish a TLS session to the server `www.esecurity.ch`.

Table C.2
The Server_Name Extension of the CLIENTHELLO Message (25 Bytes)

Bytes	#	Description
0x0000	2	Type of the extension (0 standing for `server_name`)
0x0015	2	Length of the extension (21)
0x0013	2	Length of the server name list (19)
0x00	1	Type of the server name (0 standing for `host_name`)
0x0010	2	Length of the server name (16)
0x77...68	16	ASCII encoding of the server name, i.e., `www.esecurity.ch` 77 77 77 2e 65 73 65 63 75 72 69 74 79 2e 63 68

The next two extensions, i.e., `extended_master_secret` and `renegotiation_info`, are to provide secure renegotiation and to mitigate the renegotiation and triple handshake attacks as outlined in Section 3.8.1.

DRAFT VERSION

- The `extended_master_secret` extension comprises 4 bytes that are summarized in Table C.3. According to RFC 7627, this extension is used to provide protection against the triple handshake attack. The client uses the extension to signal to the server that it actually supports the respective protection mechanism.

Table C.3
The Extended_Master_Secret Extension of the CLIENTHELLO Message (4 Bytes)

Bytes	#	Description
0x0017	2	Type of the extension (23 standing for `extended_master_secret`)
0x0000	2	Length of the extension (0)

- The `renegotiation_info` extension comprises 5 bytes that are summarized in Table C.4. According to RFC 5746, this extension is used to provide protection against the original renegotiation attack. Again, the client uses this extension to signal to the server that it it actually supports the respective protection mechanism.

Table C.4
The Renegotiation_Info Extension of the CLIENTHELLO Message (5 Bytes)

Bytes	#	Description
0xff01	2	Type of the extension (65281 standing for `renegotiation_info`)
0x0001	2	Length of the extension (1)
0x00	1	Length of the renegotiation info extension (0)

Both extensions are empty and do not include any additional data. In either case, the server is going to tell the client in the SERVERHELLO message whether it also supports the machanism (if it supports the mechanism, then the respective extension is also included in the SERVERHELLO message).

Using the following pair of related extensions, i.e., `elliptic_curves` and `ec_point_formats`, the client signals to the server that it supports ECC as outlined in Section 3.4.1.10:

- The `elliptic_curves` extension comprises 12 bytes that are summarized in Table C.5. Using this extension, the client tells the server that its supports the three elliptic curves secp256r1, secp384r1, and secp521r1 that are specified in RFC 4492. According to Table C.5, every curve is referenced with a 2-byte code.

DRAFT VERSION

Table C.5
The Elliptic_Curves Extension of the CLIENTHELLO Message (12 Bytes)

Bytes	#	Description
0x000a	2	Type of the extension (10 standing for `elliptic_curves`)
0x0008	2	Length of the extension (8)
0x0006	2	Length of the elliptic curve codes (6)
0x0017	2	Code of the elliptic curve secp256r1 (23)
0x0018	2	Code of the elliptic curve secp384r1 (24)
0x0019	2	Code of the elliptic curve secp521r1 (25)

- In addition, the `ec_point_formats` extension comprises 6 bytes that are summarized in Table C.6. Using this extension, the client tells the server that it does not compress any data related to ECC, and hence that it uses uncompressed format. As mentioned before, this is the usual behavior for most clients in use today.

Table C.6
The EC_Point_Formats Extension of the CLIENTHELLO Message (6 Bytes)

Bytes	#	Description
0x000b	2	Type of the extension (11 standing for `ec_point_formats`)
0x0002	2	Length of the extension (2)
0x01	1	Length of the point formats (1)
0x00	1	Uncompressed format (0)

The next extension, i.e., `session_ticket`, comprises 4 bytes that are summarized in Table C.7. It is used to signal to the server that the client supports session tickets as specified in RFC 5077 and explained in Section 3.4.1.18. If the server also supports session tickets and such tickets can hence be used, then they are transmitted in subsequent handshake messages (see below).

Table C.7
The Session_Ticket Extension of the CLIENTHELLO Message (4 Bytes)

Bytes	#	Description
0x0023	2	Type of the extension (35 standing for `session_ticket`)
0x0000	2	Length of the extension (0)

As mentioned in Section 3.4.1.14, NPN was a predecessor extension of ALPN (mainly used for SPDY). Support for NPN (type 13172) is signaled by the

DRAFT VERSION

next_protocol_negotiation extension that comprises 4 bytes as summarized in Table C.8, and support for ALPN (type 16) is signaled by the application_layer_protocol_negotiation extension that comprises 18 bytes as summarized in Table C.9 and specified in RFC 7301. In this table, the last 4 lines refer to the supported protocols 0x6832 standing for h2 in ASCII (and hence HTTP/2 over TLS) and 0x687474702f312e31 standing for http/1.1 (and hence HTTP/1.1 over TLS). The IANA provides a comprehensive register of all ALPN protocol IDs in use today.[1]

Table C.8
The Next_Protocol_Negotiation Extension of the CLIENTHELLO Message (4 Bytes)

Bytes	#	Description
0x3374	2	Type of the extension (13172 standing for next_protocol_negotiation)
0x0000	2	Length of the extension (0)

Table C.9
The Application_Layer_Protocol_Negotiation Extension of the CLIENTHELLO Message (18 Bytes)

Bytes	#	Description
0x0010	2	Type of the extension (16)
0x000e	2	Length of the extension (14)
0x000c	2	Length of the ALPN extension (12)
0x02	1	Length of the ALPN string (2)
0x6832	2	ALPN next protocol (h2 in ASCII)
0x08	1	Length of the ALPN string (8)
0x687474702f312e31	8	ALPN next protocol (http/1.1 in ASCII)

The status_request extension that comes next comprises 9 bytes that are summarized in Table C.10. As explained in Section 3.4.1.6, this extension asks for OCSP stapling supported by the server. In this example, the client does not ask for OCSP responses for multiple servers; otherwise, the status_request_v2 extension (value 17) would have to be invoked.

Finally, the signature_algorithms (also known as supported_signature_algorithms) extension comprises 28 bytes that are summarized in Table C.11. The client uses this extension to tell to the server what hash and signature algorithms it actually supports. Referring to Table C.11, these algorithms include SHA-1, SHA256, SHA384, and SHA512 for hashing, and RSA, DSA, and ECDSA

1 http://www.iana.org/assignments/tls-extensiontype-values/tls-extensiontype-values.xhtml#alpn-protocol-ids.

DRAFT VERSION

Table C.10
The Status_Request Extension of the CLIENTHELLO Message (9 Bytes)

Bytes	#	Description
0x0005	2	Type of the extension (5 standing for `status_request`)
0x0005	2	Length of the extension (5)
0x01	1	Certificate status type (standing for OCSP)
0x0000	1	Length of the responder ID list (0)
0x0000	2	Length of the request extensions

for signing. Note, however, that the combination of SHA512 and DSA, i.e., 0x0602, is missing in this list (in spite of the fact that this combination is defined and can theoretically be used in the field).

Table C.11
The Signature_Algorithms Extension of the CLIENTHELLO Message (28 Bytes)

Bytes	#	Description
0x000d	2	Type of the extension (13 standing for `signature_algorithms`)
0x0018	2	Length of the extension (24)
0x0016	2	Length of the signature hash algorithms (22)
0x0401	2	SHA256 (4) and RSA (1)
0x0501	2	SHA384 (5) and RSA (1)
0x0601	2	SHA512 (6) and RSA (1)
0x0201	2	SHA-1 (2) and RSA (1)
0x0403	2	SHA256 (4) and ECDSA (3)
0x0503	2	SHA384 (5) and ECDSA (3)
0x0603	2	SHA512 (6) and ECDSA (3)
0x0203	2	SHA-1 (2) and ECDSA (3)
0x0502	2	SHA384 (5) and DSA (2)
0x0402	2	SHA256 (4) and DSA (2)
0x0202	2	SHA-1 (2) and DSA (2)

After having received the CLIENTHELLO message, the server responds with a series of TLS handshake messages. The first of these messages is a SERVERHELLO message that is packaged in a TLS record and looks as follows:

```
16 03 03 00 41 02 00 00   3d 03 03 f8 7a 56 de 47
c4 fb 13 64 a6 a0 8f 18   91 46 38 5e a8 8d 68 0c
a9 c6 23 a5 58 5f 6b 53   99 98 22 00 c0 30 00 00
15 00 00 00 00 ff 01 00   01 00 00 0b 00 04 03 00
01 02 00 23 00 00
```

DRAFT VERSION

Again, the first 5 bytes refer to the TLS record header that comprises a 1-byte field referring to the content type (0x16 standing for a handshake message), a 2-byte field referring to the TLS version (0x0303 standing for TLS 1.2),[2] and a 2-byte field referring to the length of the record (0x0041 standing for 65 bytes). So the subsequent 65 bytes comprise the TLS record fragment that includes the SERVERHELLO message.

The SERVERHELLO message starts with a byte that refers to the type of the handshake message (0x02 standing for SERVERHELLO), 3 bytes that refer to the byte length of the message (0x00003d standing for 61), and 2 bytes that refer to the TLS version (0x0303 standing for TLS 1.2). The following 32 bytes, i.e., from 0xf87a to 0x9822, refer to the random value chosen by the server (note again that the first 4 bytes represent timing information). Afterwards, 0x00 refers to the length of the session ID (meaning that no session ID is transferred), 0xc030 refers to the cipher suite selected by the server (i.e., TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384), 0x00 refers to the compression method null, and 0x0015 refers to the byte length of the four TLS extensions that follow. They are summarized in Tables C.12–C.15. They basically inform the client that the server also supports the `server_name`, `renegotiation_info`, `ec_point_formats`, and `session_ticket` extensions, and hence that these extensions can be used in what follows. The other extensions may not be supported by the server, such as, for example, the `extended_master_secret` extension, or may directly be used without any further notification, such as, for example, the `elliptic_curves` extension.

Table C.12
The Server_Name Extension of the SERVERHELLO Message (4 Bytes)

Bytes	#	Description
0x0000	2	Type of the extension (0 standing for `server_name`)
0x0000	2	Length of the extension (0)

Table C.13
The Renegotiation_Info Extension of the SERVERHELLO Message (5 Bytes)

Bytes	#	Description
0xff01	2	Type of the extension (65281 standing for `renegotiation_info`)
0x0001	2	Length of the extension (1)
0x00	1	Length of the renegotiation info extension (0)

2 Here, the server signals to the client that it supports TLS 1.2 (instead of "only" TLS 1.0).

DRAFT VERSION

Table C.14
The EC_Point_Format Extension of the SERVERHELLO Message (8 Bytes)

Bytes	#	Description
0x000b	2	Type of the extension (11 standing for `ec_point_formats`)
0x0002	2	Length of the extension (2)
0x03	1	Length of the point formats (3)
0x00	1	Uncompressed point format (0)
0x01	1	ANSI X9.62 compressed prime point format (1)
0x02	1	ANSI X9.62 compressed char2 point format (2)

Table C.15
The Session_Ticket Extension of the SERVERHELLO Message (4 Bytes)

Bytes	#	Description
0x0023	2	Type of the extension (35 standing for `session_ticket`)
0x0000	2	Length of the extension (0)

According to the TLS handshake protocol, the next messages that are sent from the server to the client are the CERTIFICATE, the SERVERKEYEXCHANGE,[3] and the SERVERHELLODONE message.

The CERTIFICATE message is very large and not fully depicted here. It starts with the following 32 bytes (and comprises 2,443 additional bytes):

```
16 03 03 09 ab 0b 00 09   a7 00 09 a4 00 05 08 30
82 05 04 30 82 03 ec a0   03 02 01 02 02 12 03 99
...
```

Again, the TLS record header comprises 0x16 referring to the handshake message type, 0x0303 referring to TLS 1.2, and 0x09ab referring to 2475 that represents the total byte length of the record. The CERTIFICATE message then starts with 0x0b referring to the message type (11 standing for CERTIFICATE), 0x0009a7 referring to the message length (2471 bytes), 0x0009a4 referring to the byte length of all certificates in the certificate chain (2468 bytes), and 0x000508 referring to the byte length of the first certificate in the chain (1288 bytes). This certificate starts with the byte sequence 0x30, 0x82, and 0x05. If it is decoded, then it can be recognized that the certificate is issued for `www.esecurity.ch`. The second half of the CERTIFICATE message comprises the certificate for the issuer of the certificate for

3 This message is required, because the server has selected the cipher suite TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384 that comprises a Diffie-Hellman key exchange.

DRAFT VERSION

`www.esecurity.ch`. This certificate is 1174 bytes long and has been issued by the Let's Encrypt Authority X3. This CA, in turn, has been certfied by the DST Root CA X3 that serves as the root CA for Let's Encrypt and is preinstalled in the latest releases of the Firefox browser.

After the CERTIFICATE message, the server sends the SERVERKEYEXCHANGE message packaged in a TLS record to the client. The respective TLS record looks as follows:

```
16 03 03 01 4d 0c 00 01   49 03 00 17 41 04 12 4c
f0 6c a9 99 5a 2c 9b c7   8c 5a d3 03 02 ef 12 f6
57 d0 63 d8 b6 d6 0d 54   4e 24 61 2a 38 1f 04 51
5c c1 a1 fb 92 1a 43 ba   4e 9a 11 21 2c f7 e7 60
b7 9f 50 3c c4 37 c3 b3   e1 9a 64 26 11 69 04 01
01 00 08 78 50 6b c4 a3   63 58 32 62 aa 06 a3 22
34 1c 56 f9 02 f1 f2 b9   64 ea 54 c7 2d a3 5e 9d
45 dc 77 25 e7 b8 69 b5   96 59 ce d2 e8 fa 75 86
b0 49 d2 00 78 91 99 35   38 10 05 f8 9b da f7 21
8e c2 10 bd e7 62 23 ab   42 b2 8c d5 f6 bd fe 39
d0 6e 6a 40 5a bd 34 32   2f c9 2d 9e d5 86 05 bc
71 dd a3 a9 f2 64 1e cb   e9 16 6e 07 7f 2a ac 74
cc 0e ce 49 96 06 e6 ab   54 46 f1 28 23 c5 82 fd
a8 fb 72 0c c3 9e f5 54   57 b7 1d 69 83 8b 1e 78
16 e8 a3 e3 ab 96 7f 26   db 40 d1 86 76 f3 ed 5c
9a 72 65 e7 ab d1 81 55   2b 9a 62 15 6e 10 10 9b
08 9e 6c eb 3f 27 e5 4f   f7 0a ce ab 4e 7e 34 42
10 80 b1 80 86 86 4f b3   ed f8 8c 82 8a 68 7b 8a
d9 8d 83 3d 58 55 82 09   50 a8 5a 83 63 27 d7 e0
59 ab 87 79 1f 27 e2 79   df cd 61 ee 0f d5 ab e5
6d a3 95 15 48 8f 7d 12   22 6c 3a 22 9e 56 14 28
36 6b
```

The first 5 bytes 0x160303014d refer to the TLS record header, and the subsequent 4 bytes 0x0c000149 refer to the header of the SERVERKEYEXCHANGE message that is packaged in the record. The record length is 333 (0x014d) bytes, whereas the message length is 333 - 4 = 329 (0x000149) bytes. The first byte of the message, i.e., 0x0c, refers to the message type (12) that is standing for SERVERKEYEXCHANGE. The cipher suite is TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384, so an ECC-based ephemeral Diffie-Hellman key exchange needs to be performed. This means that the SERVERKEYEXCHANGE message is used to convey the server's Diffie-Hellman parameters to the client. The first byte after the message header, i.e., 0x03, refers to the curve type `named_curve`, and the subsequent two bytes,

DRAFT VERSION

i.e., 0x0017, refer to the first named curve secp256r1 that was originally proposed by the client. In the following byte 0x41 (65 in decimal notation), the server indicates the byte length of its public Diffie-Hellman parameter. So the subsequent 65 bytes, ranging from 0x0412 to 0x1169, refer to this parameter. In the following two pair of bytes, the server specifies the signature hash algorithm (0x0401 standing for SHA256 and RSA) and the length of the signature (0x0100 standing for 256). Consequently, the final 256 bytes, ranging from 0x0878 to 0x366b, refer to an RSA signature for the client's and server's random numbers, as well as the server's public Diffie-Hellman parameter that is included in the SERVERKEYEXCHANGE message. The client can extract and verify the RSA public key from the CERTIFICATE message, and use this key to verify the signature for the server's public Diffie-Hellman parameter accordingly.

Afterwards, the server finishes its flight by sending a SERVERHELLODONE message to the client. The respective TLS record comprises only 9 bytes that look as follows:

```
16 03 03 00 04 0e 00 00 00
```

As usual, the first 5 bytes refer to the record header. The last two bytes of this header, i.e., 0x0004, refer to the length of the fragment. This fragment, in turn, only comprises a message header. The first byte of this header, i.e., 0x0e, refers to the message type (14 standing for SERVERHELLODONE), and the subsequent 3 zero bytes refer to the length of the message. So the message comprises no data.

It is now up to the client to send a CLIENTKEYEXCHANGE message to the server in a TLS record of its own, and to provide its public Diffie-Hellman parameter accordingly. The respective TLS record looks as follows:

```
16 03 03 00 46 10 00 00   42 41 04 f5 2f 62 a6 ed
cb dd 00 4c b5 18 19 39   40 8f 28 40 44 d0 13 28
45 3a 44 79 cc 70 a3 38   62 1d f3 14 4a eb 25 b5
80 53 07 ee ed d4 8b 00   38 3b 56 b5 fb fd 43 6f
ee 4a 78 e9 85 04 67 45   bf 7e e7
```

In this record header, 0x0046 refers to the length of the fragment (70 bytes) that comprises the CLIENTKEYEXCHANGE message. The message starts with the byte 0x10 that refers to the type of the message (16 standing for CLIENTKEYEXCHANGE) and the 3 bytes 0x000042 that refer to the remaining length of the message (66 bytes). The byte 0x41 then refers to the length of the client's public Diffie-Hellman parameter that follows (65), and hence the remaining 65 bytes from 0x04f5 to 0x7ee7 refer to this parameter. Using the Diffie-Hellman parameters, the ECDHE key exchange can now be performed on either side of the session, and the client finishes the handshake by sending a CHANGECIPHERSPEC message to the server. The respective TLS record looks as follows:

DRAFT VERSION

```
14 03 03 00 01 01
```

The record consists of a TLS header and a frament with only one byte. In the header, 0x14 refers to the content type of the record (20 standing for CHANGECIPHERSPEC), 0x0303 refers to TLS 1.2, and 0x0001 refers to the length of the message (1). So the TLS message comprises a single byte, namely 0x01 referring to a CHANGECIPHERSPEC message.

Immediately following a CHANGECIPHERSPEC message, the client sends a FINISHED message to the server. Packaged in a TLS record, this message looks as follows:

```
16 03 03 00 28 00 00 00   00 00 00 00 00 28 0c c7
48 ac 1f 1e 5d 87 9e 67   3f 6d 05 24 bb 22 ea 59
7c e2 de 9e 8a 57 f8 ba   8e b5 77 a3 75
```

The first 5 bytes refer to the record header, of which 0x0028 refers to the length of the fragment (40 bytes). This fragment comprises the FINISHED message. Because it is the first message protected under the newly established keys, it is somehow more difficult to decode and understand. In essence, the 40 bytes consist of the following three components:

- An 8-byte nonce that is not encrypted. According to RFC 5116 and RFC 5288, AES-GCM encryption uses a 12-byte IV that consists of this nonce (that is explicit and transmitted in the clear) and a 4-byte salt (that is implicit and not transmitted). Because this is an initial handshake, the nonce that is used here consists of 8 zero bytes.
- A 16-byte ciphertext from 0x280c to 24bb that refers to the FINISHED message in encrypted form. The message consists of a 4-byte header and a 12-byte data field. As usual, the header includes a message type (0x14 or 20 standing for FINISHED) and a length field (0x00000c standing for 12 bytes). The data field includes the `verify_data` field that is computed as the result of the TLS PRF over the master secret, a label,[4] and the hash value for the handshake messages exchanged so far. The entire message represents a single AES block.
- A 16-byte MAC from 0x22ea to a375 that authenticates the FINISHED message.

The FINISHED message concludes the flight in which the client may send handshake messages to the server. Afterwards, it is again the server's turn. Because the client and server both provide support for the `session_ticket` extension,

4 If the client sends the FINISHED message, then the label is "client finished." Otherwise, i.e., if the server sends the FINISHED message, then the label is "server finished."

DRAFT VERSION

the server now sends a NEWSESSIONTICKET message to the client (cf. Figure 3.6). This message is packeted in a TLS record that looks as follows:

```
16 03 03 00 da 04 00 00   d6 00 00 01 2c 00 d0 e9
83 1f 96 68 12 d0 75 eb   ba 57 75 93 ec b6 41 48
dc 14 fc 56 3a 34 8a 75   2c 39 9b 56 c8 00 5e 8b
23 fc e0 83 18 ff 0a 35   0c b1 48 75 34 d4 e3 62
d2 ec 20 a1 50 f6 e2 cc   09 dc 4b 8b 38 69 40 61
fc e1 c4 0b 36 2d 59 d1   0c 1a 8a 6b ef 78 62 1c
e1 15 04 3b 8a 18 63 11   6f b9 03 45 37 67 02 c7
95 42 0d 0b 92 f8 c5 9f   75 e9 60 8c 70 e7 c7 d3
9d c9 1a 1d 48 a8 7b cd   5b 90 51 79 36 85 59 c1
d7 40 9a f9 b0 2e 58 e2   67 f5 45 36 12 19 d3 f7
16 4a 85 58 62 03 f2 cf   09 44 02 d0 4a 89 b6 62
53 4c 08 81 90 21 57 e8   cb 76 ba 1c 54 f6 c8 b4
8d da 11 04 5b 1c b7 dc   36 df 7e 02 88 38 89 62
b7 ed 3e ef 2b aa 81 9c   5b d9 75 b2 71 7a b2
```

The TLS record header specifies a fragment length of 0x00da (218) bytes. This fragment comprises the NEWSESSIONTICKET message that starts with a byte 0x04 referring to the handshake type (4 standing for NEWSESSIONTICKET) and a 3-byte length 0x0000d6 (standing for 214 bytes). The session ticket itself has 3 fields:

- The first 4 bytes 0x0000012c refer to a session ticket lifetime of 300 (300 seconds)
- The second 2 bytes 0x00d0 refer to the length of the session ticket (208 bytes).
- The remaining 208 bytes from 0xe983 to 0x7ab2 refer to the session ticket. In our example, this is by far the largest part of the NEWSESSIONTICKET message.

It is not outlined here, but the next time the client tried to resume a session, it would send this session in its CLIENTHELLO message.

Finally, the server has to send a CHANGECIPHERSPEC and a subsequent FINISHED message to the client to finish the handshake. The server's CHANGECIPHERSPEC message is identical to the client's one. It looks as follows:

```
14 03 03 00 01 01
```

The FINISHED message, in turn, in very similar to the FINISHED message the client sent to the server. It looks as follows:

```
16 03 03 00 28 fd ca 2c   10 a4 7f 0a 7c 95 2f ef
```

DRAFT VERSION

```
79 02 f5 70 44 42 96 0a   d4 40 41 d1 6b 93 f8 2a
24 3a 35 13 42 94 0a 10   34 04 02 9d 27
```

Again, the first 5 bytes refer to the record header, of which 0x0028 refers to the fragment length (40 bytes). This fragment comprises the FINISHED message that is again protected under the newly established keys. The meaning of the fields are the same as before. So the first 8 bytes refer to the explicit nonce in the clear (which does not only contain zero bytes), the next 16 bytes refer to the encrypted message, and the remaining 16 bytes refer to a MAC for this message.

After the client and server have exchanged their FINISHED messages, the TLS 1.2 session is established and application data can now be transmitted in cryptographically protected form. The first 32 bytes of a TLS record that comprises application data and is sent from the client to the server looks as follows:

```
17 03 03 01 6e 00 00 00   00 00 00 00 01 98 36 6d
e3 c0 93 91 61 26 69 d3   27 e4 5b e4 6d e5 b3 40
...
```

Here, the byte 0x17 refers to the content type of the record (23 standing for application data), 0x0303 refers to TLS 1.2, 0x016e refers to the length of the message that is packeted in the record. This message is encrypted and authenticated. It starts with an 8-byte nonce that is incremented by 1 from to the nonce used previously. The last 16 bytes refer to a 16-byte MAC, and the bytes in between refer to application data encrypted with AES-GCM.

DRAFT VERSION