|  |
| --- |
| Input Contribution |
| Meeting ID\* | TST#32 |
| Title:\* | Input on “Authentication” procedures for the Security Developer’s Guide TR-0038 |
| Source:\* | Wolfgang Granzow, Qualcomm, wgranzow@qti.qualcomm.comPhil Hawkes, Qualcomm, phawkes@qti.qualcomm.com |
| Uploaded Date:\* | 2017-11-14 |
| Document(s) Impacted\* | TR-0038v0\_2\_1 |
| Intended purpose ofdocument:\* | [x]  Decision[x]  Discussion[ ]  Information[ ]  Other <specify> |
| Decision requested or recommendation:\* | Include the text proposed in this contribution into TR-0038v0.2.1.  |
| Template Version:23 February 2015 (Do not modify) |

**oneM2M Notice**

The document to which this cover statement is attached is submitted to oneM2M. Participation in, or attendance at, any activity of oneM2M, constitutes acceptance of and agreement to be bound by terms of the Working Procedures and the Partnership Agreement, including the Intellectual Property Rights (IPR) Principles Governing oneM2M Work found in Annex 1 of the Partnership Agreement.

**Introduction**

This contribution proposes updates and new text for TR-0038.

* editorial corrections to existing section 7.1.2 and Annex A.2
* new text for clause 7.1.3 Certificate-based SAE between Home Gateway and IN-CSE
* new text for clause 7.1.4 MAF-based SAE between Smartphone and IN-CSE
* new Annex A.3 Certificate-Based Security Association Establishment
* new Annex A.4 MAF-based Security Association Establishment
* new Annex B Generation of Certificates

*======== start of change 1 =============================*

7.1.2 Provisioned Symmetric Key SAE between the Locks and the Home Gateway

In this example it is assumed that authentication between the Locks (ADN-AE1 and ADN-AE2) and the Home Gateway (MN-CSE) is performed using provisioned keys (Kpsa) and key identifiers (KpsaID).

**Configuration of ADN-AE1 and ADN-AE2:**

* The AEs are configured with the set of allowed TLS ciphersuites when using TLS-PSK as defined in clause 10.2.2 of TS-0003 [i.4]. The set of ciphersuites includes TLS\_PSK\_WITH\_AES\_128\_CBC\_SHA256.
* The AE is assumed to be configured with the CSE-ID of the Home Gateway which is a unique identifier within the M2M-SPs domain. The CSE-ID value is assumed as mn-cse-123456.
* The AE is assumed to be configured with a pair of credentials (psk, psk\_identity) associated with the CSE-ID. An example of credential configuration is given in Table 7.1.2-1. The length of the keys Kpsa is not mandated by TS-0003 [i.4] and left to implementation. In this example the key length of 8 bytes (64 bits) is chosen. The key identifiers comply with the format specified in clause 10.5 of TS-0003 [i.4].

**Table 7.1.2-1: Example Credentials configured on ADN-AE1 and ADN-AE2**

|  |  |  |
| --- | --- | --- |
| **Entity** | **Kpsa (hex format)** | **KpsaID** |
| ADN-AE1 | 1a2b3c4d5e6f7a8b | AE123456789012-Lock@in.provider.com |
| ADN-AE2 | 12345678abcdefab | AE123456789015-Lock@in.provider.com |

**Configuration of MN-CSE (Home Gateway):**

* The MN-CSE is configured with the set of allowed TLS ciphersuites when using TLS-PSK as defined in clause 10.2.2 of TS-0003 [i.4]. The set of ciphersuites includes TLS\_PSK\_WITH\_AES\_128\_CBC\_SHA256.
* The MN-CSE is assumed to have a psk-lookup-table with columns for (client identity, psk, psk\_identity), such that when a TLS client provides a particular psk\_identity, then the MN-CSE uses the corresponding psk for establishing a TLS session, and the client identity is associated with the established TLS session. This needs to be integrated to the TLS server. Table 7.1.2-2 shows an example of credentials configured on the Home Gateway to serve ADN-AE1 and ADN-AE2, containing AE-ID, KpsaID, Kpsa. A new row would need to be added to this table for each additional AE allowed to register to the MN-CSE by using TLS\_PSK.

NOTE: Some open source libraries, e.g. OpenSSL, do not provide a psk-lookup-table, but do indicate a spot in the source code where a psk-lookup could be implemented. The psk-look-up-table values could then be provided in a configuration file.

**Table 7.1.2-2: Credentials configured on MN-CSE**

|  |  |  |
| --- | --- | --- |
| **AE-ID** | **Kpsa (hex format)** | **KpsaID** |
| Clock-AE1 | 1a2b3c4d5e6f7a8b | AE123456789012-Lock@in.provider.com |
| Clock-AE2 | 12345678abcdefab | AE123456789015-Lock@in.provider.com |

**Operation of ADN-AE1 and ADN-AE2**

When the AE is triggered to establish a TLS-PSK session with the MN-CSE using some pair (Kpsa, KpsaID), the following should occur automatically based on the AE’s configuration:

* AE’s TLS Client is triggered to perform a TLS-PSK handshake with the TLS values (psk, psk\_identity) set to the values of (Kpsa, KpsaID), and with the configured list of TLS ciphersuites.
* On completion of the TLS handshake, the AE associates the established TLS session with the MN-CSE’s CSE-ID.

**Operation of MN-CSE**

The MN-CSE’ TLS Server is listening on the TLS Server port and the following should occur automatically based on the MN-CSE’s configuration:

* A TLS handshake is started at the MN-CSE TLS Server on receiving a TLS handshake Client\_Hello message. In the case of the AE, this includes the list of TLS-PSK ciphersuites supported by the AE for use with the MN-CSE. The MN-CSE will select a ciphersuite that is also in its configured list.
* A later TLS handshake message will include the psk\_identity element set to KpsaID.
* The MN-CSE’s TLS Server looks up the psk-lookup-table using KpsaID as an index, and retrieves the AE’s (AE-ID, Kpsa). If AE-ID is not available, then the MN-CSE may query the node’s <serviceSubscribedAppRule> resource.
* The MN-CSE’s TLS client continues the TLS handshake with the TLS value psk set to the value of Kpsa.
* On completion of the TLS handshake, the MN-CSE associates the established TLS session with the AE’s AE-ID.

Annex A provides details for implementing the TLS handshake procedure.

7.1.3 Certificate-based SAE between Home Gateway and IN-CSE

In this example, it is assumed that authentication between the Home Gateway (MN-CSE) and the IN-CSE is performed using CSE-ID certificates compliant with clause 10.1 of TS-0003 [i.4], which are signed by a Certification Authority (CA). The production of suitable certificates is described in Annex B.

**Configuration of MN-CSE:**

The MN-CSE is configured with the set of allowed TLS ciphersuites when using certificates as defined in clause 10.2.3 of TS-0003 [i.4]. The set of ciphersuites includes TLS\_ECDHE\_ECDSA\_WITH\_AES\_128\_CBC\_SHA256.

The MN-CSE is assumed to be configured with a CSE-ID certificate which includes its own CSE-ID in the Subject Alternative Name (subjectAltName) field (“DNS:my.example\_m2mprovider.org/mn-cse-123456”). The CSE-ID certificate is signed by a root CA certificate (in the considered example).

Table 7.1.3-1: Example credentials configured on MN-CSE

|  |  |  |  |
| --- | --- | --- | --- |
| **Entity** | **Entity-ID** | **private key file** | **certificate file** |
| MN-CSE | mn-cse-123456 | mn\_cse\_key.pem | 02.pem |

**Configuration of IN-CSE:**

The IN-CSE is configured with the set of allowed TLS ciphersuites when using certificates as defined in clause 10.2.2 of TS-0003 [i.4]. The set of ciphersuites includes TLS\_ECDHE\_ECDSA\_WITH\_AES\_128\_CBC\_SHA256.

The IN-CSE is assumed to be configured with a CSE-ID certificate which includes its own CSE-ID in the Subject Alternative Name (subjectAltName) field (“DNS:my.example\_m2mprovider.org/in-cse”). The CSE-ID certificate is signed by a root CA certificate. Acceptable CA certificates should be stored by the IN-CSE in a certificate store.

Table 7.1.3-2: Example credentials configured on IN-CSE

|  |  |  |  |
| --- | --- | --- | --- |
| **Entity** | **Entity-ID** | **private key file** | **certificate file** |
| IN-CSE | in-cse | in\_cse\_key.pem | 01.pem |

**Operation of MN-CSE**

When the MN-CSE is triggered to establish a TLS session with the IN-CSE, the following should occur automatically based on the MN-CSE’s configuration:

MN-CSE’s TLS Client is triggered to perform a TLS handshake indicating its configured list of TLS ciphersuites and providing its MN-CSE certificate upon request of the TLS server to the IN-CSE.

The MN-CSE verifies the certificate (chain) received from the IN-CSE by validating the signature(s) and by verifying that the root certificate can be trusted. Furthermore, the MN-CSE checks if the CSE-ID included in the subjectAltName field of the IN-CSEs certificate matches its configured IN-CSE ID.

On completion of the TLS handshake, the MN-CSE associates the established TLS session with the IN-CSE’s CSE-ID.

**Operation of IN-CSE**

The IN-CSE’ TLS Server is listening on the TLS Server port and the following should occur automatically based on the IN-CSE’s configuration:

A TLS handshake is started at the IN-CSE TLS Server on receiving a TLS handshake Client\_Hello message. In the case of the MN-CSE, this includes the list of TLS ciphersuites supported by the MN-CSE for use with the IN-CSE. The IN-CSE will select a ciphersuite that is also in its configured list.

The IN-CSE’s TLS Server is configured

to send its own certificate and (optional) certificate chain in a Certificate TLS handshake message

to request the certificate from the TLS client in a Certificate Request TLS handshake message and to validate this certificate

to check the CSE-ID of the MN-CSE included in the MN-CSE’s certificate. If this CSE-ID is not available, then the IN-CSE obtains it from the node’s <serviceSubscribedAppRule> resource.

On completion of the TLS handshake, the IN-CSE associates the established TLS session with the MN-CSE’s CSE-ID.

7.1.4 MAF-based SAE between Smartphone and IN-CSE

In this example, we consider the case where the AE implemented on a smartphone registers to the IN-CSE using MAF-based SAE.

We assume that the MAF client, associated with ADN-AE3 and implemented on the smartphone, is configured to use certificate-based SAE when communicating with the MAF. The MAF Client of the IN-CSE is assumed to be already registered with the MAF. The security association between AE1 and the IN-CSE is then established as illustrated in figure 7.1.4-1 with the steps described below. The communication between MAF clients and the MAF is assumed to comply with the MAF interface specification TS-0032 [i.10], where HTTP is used as binding protocol. JSON serialization of primitives is employed.



Figure 7.1.4-1: MAF-Based Security Association Establishment

1. A security association between the MAF client and the MAF is established. This procedure is the same as described in clause 7.1.4 and Annex A.3. In this example we assume that keying material to be used later on in the security association between ADN-AE3 and IN-CSE is derived at both ends using the TLS key exporter function (see clauses 8.2.2.3 and 8.3.5.3.7 of TS-0003 [i.4]). Further details of this procedure are described in Annex A.4.

 *Editor’s note: When a MAF client is associated with a single AE or CSE, an already existing AE-ID or CSE-ID certificate may be used in the TLS handshake. This would require some clarifications in TS-0003. TS-0003 currently mandates the use of a device certificate, which requires a device ID in subjectAltName.*

1. The MAF client registers to the MAF by sending a MAF client registration request as specified in clause 8.8.2.3 of TS-0003 [i.4]:

|  |  |
| --- | --- |
| **JSON serialized primitive** | **Comments** |
| {"rqp": {  "op": 1,  "to": "//myMAF.provider.org/-/", "fr": "0 2 481 1 100 3030 10011",  "rqi": "0001",  "ty": 3,  "pc": {"sec:macr": {  "et": "20181113T110000",  "adfq": "mytrustenabler.org"  }}, "rcn": 7 }} | request primitiveoperation = CREATEto = default MAFBase address from = device id of device where MAF client is installed request identifier, assigned by originatorresource type = <mafClientReg> to be createdcontent = global element name of <mafClientReg>expirationTime = 2018-11-13 11:00:00 UTCadminFQDN result content = Original Resource |

1. The MAF sends the response to the MAF client:

|  |  |
| --- | --- |
| **JSON serialized primitive** | **Comments** |
| {"rsp": { "rsc": 2001,  "rqi": "0001", "pc": {"sec:macr": { "rn": "MACR000001", "ty": 3, "ri": "macr000001", "pi": "mb01", "ct": "20171113T110000", "lt": "20171113T110000", "et": "20181113T110000", "cr": "0 2 481 1 100 3030 10011", "adfq": "mytrustenabler.org", "aski": "FF15D84E3E38D6974B0EB3E5606C85FE@myMAF.provider.org" }}}} | response primitiveresponse status code, CREATEDrequest identifiercontent=global element name <mafClientReg>resource name, assigned by MAF resource type = <mafClientReg>resource identifier, assigned by MAFparent identifier, resource id of MAFBasecreation timelast modified timeexpiration time, 1 year after creationcreator, MAF client idadminFQDN, fqdn of trust enablerkey identifier |

1. MAF key registration request as described in clause 8.8.2.7 of TS-0003 [i.4].

|  |  |
| --- | --- |
| **JSON serialized primitive** | **Comments** |
| {"rqp": { "op": 1, "to": "//myMAF.provider.org/-/macr000001", "fr": "0 2 481 1 100 3030 10011", "rqi": "0002", "ty": 5, "pc": {"sec:mkr": { "et": "20171120T110000", "adfq": "mytrustenabler.org", "suid": 11 }}, "rcn": 7}} | request primitiveoperation = CREATEto = address of <mafClientReg> parent resourcefrom = device id of MAF client (= MAF client ID) request identifier, assigned by originatorresource type = <symmKeyReg> to be createdcontent = global element name of <symmKeyReg>expiration time, 1 week after creationadminFQDN, fqdn ofd trust enablersecurity usage id = MAF-based SAEFresult content = Original Resource |

1. MAF key registration response. Note that the *keyValue* attribute is not returned to the MAF client as this key is derived from the TLS key exporter function.

|  |  |
| --- | --- |
| **JSON serialized primitive** | **Comments** |
| {"rsp": { "rsc": 2001, "rqi": "0002", "pc": {"sec:mkr": { "rn": "SK00001", "ty": 5, "ri": "FF15D84E3E38D6974B0EB3E5606C85FE", "pi": "macr000001", "ct": "20171113T110001", "lt": "20171113T110001", "et": "20171120T110001", "cr": "0 2 481 1 100 3030 10011", "adfq": "mytrustenabler.org", "suid": 11, "tgis": "//my.m2mprovider.org/in-cse" }}}} | response primitiveresponse status code, CREATEDrequest identifiercontent=global element name <symmKeyReg>resource name, assigned by MAF resource type = <symmKeyReg>resource identifier, assigned by MAF equal to relativeKeyID, see Annex A.4parent identifier, resource id of <mafClientReg>creation timelast modified timeexpiration time, 1 week after creationcreator, MAF client idadminFQDN, fqdn of trust enablersecurity usage id = MAF-based SAEFlist of target identifiers, registrar CSE idNote: key value is not returned to MAF client in this procedure |

1. Using the keying material established in step 1 the security credentials psk and psk\_identity are transferred from the MAF client to the AE (see Annex A.4 for more details).
2. PSK-based security association is established between AE3 and the IN-CSE, as described in clause 7.1.3 and Annex A.2 using psk and psk\_identity from step 6.
3. As part of step 7), the MAF client associated with the IN-CSE retrieves the PSK credential from the MAF which is identified from the fqdn-part of the psk\_identity value by means of triggering a MAF Key Retrieval procedure as specified in clause 8.8.2.8 of TS-0003 [i.4]. It is assumed that a security association between IN-CSE's MAF client and the MAF already exists prior to execution of the MAF Key Retrieval procedure.

The Key Retrieval request and response primitives are shown in the Table below:

|  |  |
| --- | --- |
| **JSON serialized primitive** | **Comments** |
| {"rqp": { "op": 2, "to": "//myMAF.provider.org/-/FF15D84E3E38D6974B0EB3E5606C85FE", "fr": "//my.m2mprovider.org/in-cse", "rqi": "ABC28F", "rcn": 7}} | request primitiveoperation = RETRIEVEto = address of <symmKeyReg> parent resource = KcIDfrom = IN-CSE identifier request identifier, assigned by originatorresult content = Original Resource |
| {"rsp": { "rsc": 2000, "rqi": "ABC28F", "pc": {"sec:mkr": { "rn": "SK00001", "ty": 5, "ri": "FF15D84E3E38D6974B0EB3E5606C85FE", "pi": "macr000001", "ct": "20171113T110001", "lt": "20171113T110001", "et": "20171120T110001", "cr": "0 2 481 1 100 3030 10011", "adfq": "mytrustenabler.org", "suid": 11, "tgis": "//my.m2mprovider.org/in-cse", "kv": "37F61D5A7FEA1E9CFD8DB76D2F8B6230130EF8A84F9F9F967DA385867984EED0" }}}} | response primitiveresponse status code, OKrequest identifiercontent=global element name <symmKeyReg>resource name, assigned by MAF resource type = <symmKeyReg>resource identifier = relative Key idparent identifier, resource id of <mafClientReg>creation timelast modified timeexpiration time, 1 week after creationcreator, MAF client idadminFQDN, fqdn of trust enablersecurity usage id = MAF-based SAEFlist of target identifiers, registrar CSE idkey value, as derived with TLS key material exporter function |

1. Encrypted messages can be exchanged between AE3 and the IN-CSE.

7.1.5 Registration upon successful SAE

*Editor’s note: this clause will provide an example of the registration procedure following successful Security Association Establishment. This procedure is independent of the SAE procedures described in clauses 7.1.2 to 7.1.4. It will also include an example of AE impersonation checking procedure*.

*======== end of change 1 =============================*

*======== start of change 2 =============================*

# Annex A:

# Security Association Establishment Message Flows

## A.1 Introduction

This Annex presents some example message flows which are useful to understand the operation of the oneM2M security establishment frameworks, to verify correct operation or to identify the cause of misbehavior.

Some details of TLS message flows and message content depend on the employed SSL/TLS implementation. Implementations of oneM2M entities will typically make use of SSL/TLS libraries to enable support of the required security functions specified in TS-0003. Examples of open source SSL/TLS libraries include *OpenSSL*, *gnuSSL* and *mbed TLS*.

Such SSL/TLS libraries implement the basic cryptographic functions and provide various utility functions such as e.g. TLS clients and servers which may be executed from a command line.

The message flows shown here have been produced using OpenSSL Version 1.1.1-dev on an Ubuntu 14.04 computer using the s\_client and s\_server utility functions, and employing Wireshark for capturing and analyzing the exchanged data packets. Note that OpenSSL Version 1.1.0 or higher is required to support the PSK ciphers defined in RFC 5989 and mandated to be used by TS-0003.

The commands given in the subsections below may be used to reproduce these flows.

## A.2 PSK-Based Security Association Establishment

A typical flow of messages and actions for a successful PSK-Based Security Association Establishment is shown in figure A.2-1. The message content described in the steps below applies to the example described in clause 7.1.2.

Subsequent to TCP connection establishment (not shown in the Figure), the following messages are exchanged between ADN-AE1 and the MN-CSE:

1. The TLS client on ADN-AE1 sends a Client Hello Handshake message which is encapsulated in a TLS Record layer frame. The record layer message includes the following fields:
2. Record layer header fields:
	* + - Content type 0x16 (Handshake)
			- Version 0x0301 (indicating TLS 1.0)
			- Length of the message (2 bytes, value depending on the message content)
3. Application data (handshake message):
	* + - Handshake Type 0x01 (Client Hello)
			- Length of the message (3 bytes, value depending on the message content)
			- Client Version 0x0303 (TLS 1.2)
			- (Client) Random (32 bytes, generated by the TLS client’s pseudo random number generator (PRNG))
			- Length of cipher suites field (value at least 1)
			- List of cipher suites supported by the client. Must include identifier for TLS\_PSK\_WITH\_AES\_128\_CBC\_SHA256 (0x00ae)
			- Extension length and Extensions (irrelevant for this example)
4. The TLS server handshake protocol responds with Server Hello and Server Hello Done messages. For the implementation employed here, each of these messages is encapsulated into a dedicated record layer frame.
5. Record layer header fields:
	* + - Content type 0x16 (Handshake)
			- Version 0x0303 (indicating TLS 1.2)
			- Length of the application data field (2 bytes, value depending on the message content)
6. Application data (“Server Hello” handshake message):
	* + - Handshake Type 0x02 (Server Hello)
			- Length of the message (3 bytes, value depending on the message content)
			- Server version 0x0303 (indicating TLS 1.2)
			- (Server) Random (32 bytes, generated by the TLS server’s PRNG)
			- Session-Id length (0x00, no session ID supplied)
			- Cipher suite selected by the server is TLS\_PSK\_WITH\_AES\_128\_CBC\_SHA256 (0x00ae)
			- Compression method (null, no compression)
			- Extension length and Extensions (irrelevant for this example)
7. Record layer header fields:
	* + - Same as in step 2.i
8. Application data (“Server Hello Done” handshake message):
	* + - Handshake type 0x0e (Server Hello Done)
			- Length of the message (0x0000, message has no content)
9. The TLS client responds with Client Key exchange, Change Cipher Spec, Finished messages. For the implementation employed here, each of these messages is encapsulated into a dedicated record layer frame.
10. Record layer header fields:
	* + - Same as in step 2.i
11. Application data (“Client Key Exchange” handshake message):
	* + - Handshake Type 0x10 (Client Key Exchange)
			- Length of the message (3 bytes, value depending on the message content)
			- PSK client parameters:
				* Identity length ( 0x00000f in this example)
				* PSK Identity (here binary equivalent of “Client\_identity”)
12. Record layer header fields:
	* + - Content type 0x14 (Change Cipher Spec)
			- Version 0x0303 (TLS 1.2)
			- Length of the message (0x0001)
13. Application data (“Change Cipher Spec” message):
	* + - Change Cipher Spec message 0x01 (1 byte)
14. Record layer header fields:
	* + - Same as in step 2.i
15. Application data (encrypted “Finished” handshake message)
	* + - Handshake type 0x14 (Finished)
			- Length of the message 0x00000c (12)
			- Verify Data (12 bytes), see RFC 5246, section 7.4.9.
16. The server retrieves Kpsa associated with the PSK Identity, computes the master secret and authenticates the client by validating Verify Data
17. The TLS server responds with New Session Ticket, Change Cipher Spec, Finished messages. For the implementation employed here, each of these messages is encapsulated into a dedicated record layer frame.
18. Record layer header fields:
	* + - Same as in step 2.i
19. Application data (“New Session Ticket” handshake message):
	* + - Handshake Type 0x04 (New Session Ticket)
			- Length of the message (3 bytes: 0x0000b6)
			- Session Ticket:
				* Lifetime Hint (4 bytes: 0x00001c20, 7200 in this example)
				* Session Ticket Length (2 bytes, 0x00b0, 176 in this example)
				* Session Ticket (176 bytes), see RFC 4507, server session state enabling session resumption
20. Record layer header fields:
	* + - Content Type 0x14 (Change Cipher Spec)
			- Version 0x0303 (TLS 1.2)
			- Length of the message (0x0001)
21. Encrypted application data (“Change Cipher Spec” message):
	* + - Change Cipher Spec message 0x01 (1 byte)
22. Record layer header fields:
	* + - Same as in step 2.i
23. Application data (encrypted “Finished” handshake message, to verify that the key exchange and authentication processes were successful):
	* + - Handshake Type 0x14 (Finished)
			- Length of the message 0x00000c (12)
			- Verify Data (12 bytes), see RFC 5246, section 7.4.9.
24. The client authenticates the server by validating Verify Data
25. Application data encrypted by the TLS record layer is exchanged between ADN-AE1 and MN-CSE



Figure A.2-1: PSK-Based Security Association Establishment

The message flow described above (excluding step 7) can be reproduced with the following commands under Linux OS using localhost IP address and port 443:

**TLS server on MN-CSE:**

$ sudo openssl s\_server -accept 443 -psk 1a2b3c4d5e6f7a8b

**TLS Client on ADN-AE1:**

$ openssl s\_client -connect 0.0.0.0:443 -psk\_identity Client\_identity \

 -psk 1a2b3c4d5e6f7a8b -cipher PSK-AES128-CBC-SHA256

NOTE: The OpenSSL s\_server utility does not support table lookup of pre-shared keys when using the option

 -psk\_identity AE123456789015-Lock@in.provider.com

as required for the example in clause 7.1.2. Therefore, the above command line for the server includes the used PSK itself. The client command line provides the PSK identity “Client\_identity” which is expected by the server for this PSK.

Note that in order to enable Wireshark to decrypt application data which has been encrypted by the TLS record layer, it must be configured as follows:

In the Wireshark configuration menu Edit -> Preferences -> Protocols -> SSL,

1. In the “Pre-Shared-Key” field, enter Kpsa, i.e. 1a2b3c4d5e6f7a8b
2. In the (Pre)-Master-Secret log filename field, enter the name of a text file which includes Client Random (32 bytes as 64 hex characters) and the Master Secret (48 bytes as 96 hex characters) as a text line as follows:

 CLIENT\_RANDOM <space> 64-characters-random <space> 96-characters-Master-Secret

The master secret is provided as log information in the terminal window, where s\_client is started. The value of Client Random can be retrieved from the Wireshark packet capture in the Client Hello handshake message.

First the data captured with Wireshark must be stored into a file. Then, after configuring Wireshark as described above, the messages in the saved data file can be decrypted by Wireshark.

*Editor’s note: relation between credential identifiers, entity identifiers and service subscription information needs to be clarified*

## A.3 Certificate-Based Security Association Establishment

Figure A.3-1 shows a typical flow of messages and actions for a successful certificate-based Security Association Establishment. The message content, i.e. the names of certificate files, private key files and CSE identifiers, described in the steps of the message flow, corresponds to the example described in clause 7.1.3.

Subsequent to TCP connection establishment (not shown in the Figure), the following messages are exchanged between ADN-AE1 and the MN-CSE:

1. The TLS client on MN-CSE sends a Client Hello Handshake message which is encapsulated in a TLS Record layer frame. The record layer message includes the following fields:
2. Record layer header fields:
	* + - Content type 0x16 (Handshake)
			- Version 0x0301 (indicating TLS 1.0)
			- Length of the message (2 bytes, value depending on the message content)
3. Application data (handshake message):
	* + - Handshake Type 0x01 (Client Hello)
			- Length of the message (3 bytes, value depending on the message content)
			- Client Version 0x0303 (TLS 1.2)
			- (Client) Random (32 bytes, generated by the TLS client’s pseudo random number generator (PRNG))
			- Length of cipher suites field
			- List of cipher suites supported by the client. This list must include TLS\_ECDHE\_ECDSA\_WITH\_AES\_128\_CBC\_SHA256 (0xc023)
			- Extension length and Extensions (includes ec\_point\_formats, eliptic\_curves, SessionTicket TLS, signature\_algorithms)
4. The TLS server handshake protocol responds with Server Hello, Certificate, Server Key Exchange, Certificate Request and Server Hello Done messages. For the implementation employed here, each of these messages is encapsulated into a dedicated record layer frame.
5. Record layer header fields:
	* + - Content type 0x16 (Handshake)
			- Version 0x0303 (indicating TLS 1.2)
			- Length of the application data field (2 bytes, value depending on the message content)
6. Application data (“Server Hello” handshake message):
	* + - Handshake Type 0x02 (Server Hello)
			- Length of the message (3 bytes, value depending on the message content)
			- Server version 0x0303 (indicating TLS 1.2)
			- (Server) Random (32 bytes, generated by the TLS server’s PRNG)
			- Session-Id length (0x00, no session ID supplied)
			- Cipher suite selected by the server, should be TLS\_PSK\_WITH\_AES\_128\_CBC\_SHA256 (0x00ae)
			- Compression method (null, no compression)
			- Extension length and Extensions (only extension types included, irrelevant for this example)
7. Record layer header fields:
	* + - Same as in step 2.i
8. Application Data (“Certificate” handshake message): includes IN-CSE certificate and the Certificate
	* + - Handshake type 0x11 (Certificate)
			- Length of the message (3 bytes, value is 1224, for the given certificates)
			- Certificate length (3 bytes)
			- Certificate (601 bytes): MN-CSE certificate
			- Certificate length 3 bytes
			- Certificate 614 bytes: IN-CSE certificate
9. Record layer header fields:
	* + - Same as in step 2.i
10. Application Data (“Server Key Exchange” handshake message):
	* + - Handshake type 0x0c (Server Key Exchange)
			- Length of the message (3 bytes)
			- EC Diffie-Hellman Server Parameters
11. Record layer header fields:
	* + - Same as in step 2.i
12. Application Data (“Certificate Request” handshake message):
	* + - Handshake type 0x0d (Certificate Request)
			- Length of the message (3 bytes)
			- Certificate Types, Signature Hash Algorithms
			- Distinguished Names, includes the issuer of the certificate
13. Record layer header fields:
	* + - Same as in step 2.i
14. Application data (“Server Hello Done” handshake message):
	* + - Handshake type 0x0e (Server Hello Done)
			- Length of the message (0x0000, message has no content)
15. The TLS client validates the certificate (chain) received from the TLS server.

The client validates the signature(s) of the certificate(s) and checks if it can trust the root certificate.

1. The TLS client responds with Certificate, Client Key exchange, Certificate Verify, Change Cipher Spec, Finished messages. For the implementation employed here, each of these messages is encapsulated into a dedicated record layer frame.
2. Record layer header fields:
	* + - Same as in step 2.i
3. Application data (“Certificate” handshake message):
	* + - Handshake Type 0x0b (Certificate)
			- Length of the message (3 bytes, value depending on the message content, 608 bytes in this example)
			- Certificates length (3 bytes, length of certificate chain, value is 605 bytes for the given certificate 02.pem)
			- Certificate length (3 bytes, value is 602 bytes for the certificate given in 02.pem)
			- Certificate (ASN.1 DER encoded binary representation of the certificate included in 02.pem)
4. Record layer header fields:
	* + - Same as in step 2.i
5. Application data (“Client Key Exchange” handshake message):
	* + - Handshake Type 0x10 (Client Key Exchange)
			- Length of the message (3 bytes, value depending on the message content)
			- PSK client parameters:
				* Identity length ( 0x00000f in this example)
				* PSK Identity (here binary equivalent of “Client\_identity”)
6. Record layer header fields:
	* + - Same as in step 2.i
7. Application data (“Certificate Verify” handshake message):
	* + - Handshake Type 0x0f (Certificate Verify)
			- Length of the message (3 bytes, value depending on the message content)
			- Signature hash algorithm (ECDSA with SHA256, Signature Length (72 bytes) and Signature of all sent or received handshake messages of the current TLS handshake, see Section 7.4.8 of RFC5246
8. Record layer header fields:
* Same as in step 2.
1. Application data (“Change Cipher Spec” message):
	* + - Change Cipher Spec message 0x01 (1 byte)
2. Record layer header fields:
	* + - Same as in step 2.i
3. Application data (encrypted “Finished” handshake message)
	* + - Handshake type 0x14 (Finished)
			- Length of the message 0x00000c (12)
			- Verify Data (12 bytes), see RFC 5246, section 7.4.9.
4. The server validates the certificate (chain) received from the client.
5. The TLS server responds with New Session Ticket, Change Cipher Spec, Finished messages. For the implementation employed here, each of these messages is encapsulated into a dedicated record layer frame.
6. Record layer header fields:
	* + - Same as in step 2.i
7. Application data (“New Session Ticket” handshake message):
	* + - Handshake Type 0x04 (New Session Ticket)
			- Length of the message (3 bytes: 0x0000b6)
			- Session Ticket:
				* Lifetime Hint (4 bytes: 0x00001c20, 7200 in this example)
				* Session Ticket Length (2 bytes, 0x00b0, 176 in this example)
				* Session Ticket (176 bytes), see RFC 4507, server session state enabling session resumption
8. Record layer header fields:
	* + - Content Type 0x14 (Change Cipher Spec)
			- Version 0x0303 (TLS 1.2)
			- Length of the message (0x0001)
9. Encrypted application data (“Change Cipher Spec” message):
	* + - Change Cipher Spec message 0x01 (1 byte)
10. Record layer header fields:
	* + - Same as in step 2.i
11. Application data (encrypted “Finished” handshake message, to verify that the key exchange and

 authentication processes were successful):

* + - * Handshake Type 0x14 (Finished)
			* Length of the message 0x00000c (12)
			* Verify Data (12 bytes), see RFC 5246, section 7.4.9.
1. The client authenticates the server by validating the Verify Data field and by matching of the CSE-ID in the subjectAltName field with its preconfigured registrar CSE-ID. Also, the server may check if the client’s MN-CSE-ID given in the subjectAltName field of the client certificate is already registered or is allowed to register to the IN-CSE (e.g. by checking if there is a <serviceSubscribedNode> resource instance which includes this MN-CSE ID.
2. Service Layer data encrypted by the TLS record layer is exchanged between MN-CSE and IN-CSE



Figure A.3-1: Certificate-Based Security Association Establishment

The message flow described above (excluding step 7) can be reproduced with the following commands under Linux OS using localhost IP address and port 443 (it is assumed that path names apply and CSE-certificates are available in the directory from where this command is issued):

**TLS server on IN-CSE:**

$ sudo openssl s\_server -accept 443 -Verify 1 -key in\_cse\_key.pem \

 -cert 01.pem -CApath ./demoCA -CAfile ./demoCA/cacert.pem

**TLS client on MN-CSE:**

$ openssl s\_client -connect 0.0.0.0:443 -key mn\_cse\_key.pem -cert 02.pem \

 -verify 1 –cipher ECDHE-ECDSA-AES128-SHA256 \

 -CApath ./demoCA -CAfile ./demoCA/cacert.pem

NOTE: CipherSuite TLS\_ECDHE\_ECDSA\_WITH\_AES\_128\_CBC\_SHA256 = {0xC0,0x23} as defined in RFC5989 is denoted ECDHE-ECDSA-AES128-SHA256 in openssl [i.9]

Note that in order to enable Wireshark to decrypt application data which has been encrypted by the TLS record layer, it must be configured as follows:

In the Wireshark configuration menu Edit -> Preferences -> Protocols -> SSL,

* In the (Pre)-Master-Secret log filename field, enter the name of a text file which includes Client Random (32 bytes as 64 hex characters) and the Master Secret (48 bytes as 96 hex characters) as a text line as follows:

 CLIENT\_RANDOM <space> 64-characters-random <space> 96-characters-Master-Secret

The master secret is provided as log information in the terminal window, where s\_client is started. The value of Client Random (comprised of GMT Time (4 bytes/8 hex chars) plus Random (28 bytes/56 hex chars)) can be retrieved from the Wireshark packet capture in the Client Hello handshake message.

## A.4 MAF-Based Security Association Establishment

In MAF-based Security Association Establishment between two oneM2M entities (i.e. AEs and CSEs) symmetric key credentials are employed which have been established with a preceding procedure on a MAF. This key establishment procedure corresponds to steps 1 to 6 in the example described in clause 7.1.4.

Step 1 of the procedure in clause 7.1.4 represents a certificate-based TLS-handshake between MAF client and MAF where in addition the keying material exporter function as defined in RFC 5705 (RFC 65705) is enabled.

The handshake message flow of this step can be produced with the following commands under Linux OS using a DNS-resolvable MAF-FQDN *myMAF.provider.org* and port 443 (it is assumed that path names apply and certificates are available in the directory from where this command is issued):

**TLS server on MAF with example FQDN *myMAF.provider.org*:**

$ sudo openssl s\_server -accept 443 -Verify 1 -key maf\_key.pem \

 -cert maf\_cert.pem -CApath ./demoCA -CAfile ./demoCA/cacert.pem \

 -keymatexport EXPORTER-oneM2M-Connection -keymatexportlen 48

**TLS client on MAF client associated with AE3:**

$ openssl s\_client -connect myMAF.provider.org:443 -key maf\_client\_key.pem \

 -cert maf\_client\_cert.pem -verify 1 –cipher ECDHE-ECDSA-AES128-SHA256\

 -keymatexport EXPORTER-oneM2M-Connection -keymatexportlen 48

At both TLS endpoints, openssl produces an output such as the following (example):

Keying material exporter:

 Label: 'EXPORTER-oneM2M-Connection'

 Length: 48 bytes

 Keying material: FF15D84E3E38D6974B0EB3E5606C85FE

 37F61D5A7FEA1E9CFD8DB76D2F8B6230

 130EF8A84F9F9F967DA385867984EED0

The value of Keying material is a 48 byte array represented as a 96-character hexadecimal string which is divided into two parts:

* upper 16 bytes (32 hex characters), denoted as Connection Key Identifier (KcID):
	+ FF15D84E3E38D6974B0EB3E5606C85FE
* lower 32 bytes (64 hex characters), denoted as M2M Secure Connection Key (Kc):
	+ 37F61D5A7FEA1E9CFD8DB76D2F8B6230130EF8A84F9F9F967DA385867984EED0

From KcID, the *Key Identifier* is derived as follows (see clause 10.3.5 of TS-0003 [i.4]):

Key Identifier = RelativeKeyID@MAF-FQDN

where RelativeKeyID = hexBinary(KcID) and MAF-FQDN is the domain name of the MAF on which the key Kc which is associated with the Key Identifier is registered. For the above example of MAF-FQDN and KcID, the Key Identifier is derived as:

hexBinary(0xFF15D84E3E38D6974B0EB3E5606C85FE) = 'FF15D84E3E38D6974B0EB3E5606C85FE'

Key Identifier: 'FF15D84E3E38D6974B0EB3E5606C85FE@myMAF.provider.org'

Note that the value of the *resourceID* attribute of <*symmKeyReg*> resources instances hosted on the MAF identified by MAF-FQDN is set to the RelativeKeyID.

The hexadecimal representation of the key Kc associated with this Key Identifier will be stored in the *keyValue* attribute of a <*symmKeyReg*> resource instance, which is created in step 4 of the message sequence given in Figure 7.1.4-1.

Annex B:
Generation of Certificates

## B.1 Introduction

This Annex describes how to generate certificates which are compliant with the requirements defined in TS-0003 [i.4].

Generation of certificates requires setting up a simple Public Key Infrastructure (PKI). It is outlined here how this can be accomplished using OpenSSL. For simplicity a root CA is setup which employs a self-signed root certificate to sign all end user’s certificates. The end users of the certificates in the present context refer to AEs or CSEs.

The private keys and certificates need to be deployed in AEs and CSEs in a secure way. Private keys require special protection on devices. They should be stored and be employed for security procedures in a secure environment. Note that these aspects are not addressed in this Annex. A simple way to protect keys is to store them in password protected files. However, for simplicity, in the following procedures this feature is not used.

Furthermore, the following conditions and conventions apply:

* all generated keys support elliptic curve Diffie-Hellman encryption (ECDHE) and elliptic curve digital signature Algorithm (ECDSA),
* all keys and certificates are generated in Privacy-Enhanced Mail (PEM) format and are stored in files with extension *.pem,*
* the described examples have been tested using OpenSSL v1.1.1-dev under a Ubuntu 14.04 LTS operating system.

Note that any addresses used in the examples shown in the present annex, e.g. in the issuer and subject fields of the generated certificates, are just arbitrary examples not applicable for real implementations.

## B.2 Setting up a root CA

When installing OpenSSL on a Linux computer, a configuration file openssl.cnf is created by default in the directory /etc/ssl.

The information in openssl.cnf defines sets of parameters which are applied by default by the openssl command line utility functions. Additional information on OpenSSL PKI and certificate generation can be found in [i.7] and [i.8].

The following section should be included into the default version of openssl.cnf to get the commands shown below and in clause B.3 to work properly:

####################################################################

[ ca ]

default\_ca = CA\_default # The default ca section

####################################################################

[ CA\_default ]

dir = ./demoCA # Where everything is kept

certs = $dir/certs # Where the issued certs are kept

crl\_dir = $dir/crl # Where the issued crl are kept

database = $dir/index.txt # database index file.

unique\_subject = no # Set to 'no' to allow creation of

 # several certificates with same subject.

new\_certs\_dir = $dir/newcerts # default place for new certs.

certificate = $dir/cacert.pem # The CA certificate

serial = $dir/serial # The current serial number

crlnumber = $dir/crlnumber # the current crl number

 # must be commented out to leave a V1 CRL

crl = $dir/crl.pem # The current CRL

private\_key = $dir/private/cakey.pem # private key of the root cert

RANDFILE = $dir/private/.rand # private random number file

 # (not used in the present example)

x509\_extensions = usr\_cert # The extensions to add to the cert

[signing\_policy]

countryName     = optional

stateOrProvinceName = optional

localityName        = optional

organizationName    = optional

organizationalUnitName = optional

commonName      = supplied

emailAddress    = optional

subjectAltName  = supplied

Create or change to some existing directory, where the tree containing private keys and certificates should originate. From this directory, execute the following commands:

$ mkdir demoCA

$ mkdir demoCA/newcerts

$ mkdir demoCA/private

$ sh -c "echo '01' > ./demoCA/serial"

$ touch ./demoCA/index.txt

These commands create the directory structure and the files which control the generation of the serial number of the certificates. The serial number of the end user certificates created by the CA will be incremented starting from 01.

## B.2 Generation of CA private key and root certificate

The command given below generates a CA key in a file cakey.pem with implicit elliptic curve parameters from the curve named secp256r1 (note that OpenSSL uses curve prime256v1 which is the same as secp256r1):

$ openssl ecparam -name secp256r1 -genkey -out cakey.pem

The command below generates a self-signed root certificate with the name cacert.pem:

$ openssl req -new -x509 -extensions v3\_ca -key cakey.pem -subj "/C=US/ST=California/O=Trusted Certificate Authority/CN=mtrusted\_ca.com/emailAddress=service@trusted\_ca.com" -out cacert.pem -days 3650

The private key and certificate files need be moved into the directories as configured in openssl.cnf:

$ mv cakey.pem demoCA/private/.

$ mv cacert.pem demoCA/.

## B.3 Generation of end user private key and certificates

This clause shows commands which generate the end user certificates which are signed by the root CA. These certificates are employed in the example described in Annex A.3 by the IN-CSE and MN-CSE. The Subject Alternative Name of these certificates include the CSE-IDs of the IN-CSE and MN-CSE, respectively.

The following commands generate the key files:

$ openssl ecparam -name secp256r1 -genkey -out in\_cse\_key.pem

$ openssl ecparam -name secp256r1 -genkey -out mn\_cse\_key.pem

The following commands generate signing requests (CSRs) for the IN-CSE and MN-CSE certificates:

$ openssl req -new -extensions SAN -key in\_cse\_key.pem -subj "/C=US/ST=California/O=MY\_M2M\_PROVIDER, Inc./CN=my.m2mprovider.org" -reqexts SAN -config <(cat /etc/ssl/openssl.cnf <(printf "[SAN]\nsubjectAltName=DNS:my.m2mprovider.org/in-cse")) -out in\_cse\_cert.csr -days 365

$ openssl req -new -extensions SAN -key mn\_cse\_key.pem -subj "/C=US/ST=California/O=MY\_M2M\_PROVIDER, Inc./CN=my.m2mprovider.org" -reqexts SAN -config <(cat /etc/ssl/openssl.cnf <(printf "[SAN]\nsubjectAltName=DNS:my.m2mprovider.org/mn-cse")) -out mn\_cse\_cert.csr -days 365

The following command generate the signed IN-CSE certificate from the CSR. This produces a certificate ./demoCA/newcerts/01.pem:

$ openssl ca -in in\_cse\_cert.csr -policy signing\_policy -config /etc/ssl/openssl.cnf -extensions SAN -config <(cat /etc/ssl/openssl.cnf <(printf "[SAN]\nsubjectAltName=DNS:my.m2mprovider.org/in-cse")) -verbose

The following command generate the signed MN-CSE certificate from the CSR. This produces a certificate ./demoCA/newcerts/02.pem:

$ openssl ca -in mn\_cse\_cert.csr -policy signing\_policy -config /etc/ssl/openssl.cnf -extensions SAN -config <(cat /etc/ssl/openssl.cnf <(printf "[SAN]\nsubjectAltName=DNS:my.m2mprovider.org/mn-cse-123456")) -verbose

The private keys and certificates would need to be deployed on the end entities (i.e. IN-CSE with CSE-ID = in-cse and MN-CSE with CSE-ID = mn-cse-123456).

For testing of certificate-based TLS-handshake as described in Annex A.3, these certificates and private keys may be copied into the directory from where the opennssl s\_server and s\_client commands given in Annex A.3 are executed.

*======== End of change 2 =============================*

*======== Start of change 3 =============================*

2.2 Informative references

*Clause 2.2 shall only contain informative references which are cited in the document itself.*

The following referenced documents are not necessary for the application of the present document but they assist the user with regard to a particular subject area.

[i.1] oneM2M Drafting Rules (http://www.onem2m.org/images/files/oneM2M-Drafting-Rules.pdf)

[i.2] oneM2M TS-0001: "Functional Architecture".

[i.3] oneM2M TS-0004: "Service Layer Core protocol Specification”.

[i.4] oneM2M TS-0003: "Security Solutions".

[i.5] oneM2M TS-0011: "Common Terminology".

[i.6] oneM2M TR-0025: "Application Developer Guide"

[i.7] Stefan H. Holek: "OpenSSL PKI Tutorial", Release v1.1, 13-Aug-2017

[i.8] Ivan Ristić: "OpenSSL Cookbook ", Version 1.1, Oct-2013

[i.9] OpenSSL User Manual, <https://www.openssl.org/docs/manmaster/man1/ciphers.html>

[i.10] oneM2M TS-0032: "MAF and MEF Interface Specification"

*======== End of change 3 =============================*