Mesh Access Connection

Purpose

The MeshAccessConnection works closely with the MeshAccessModule to provide a simple proxy for connecting to the mesh network from a device that does not need to implement the FruityMesh algorithm.

Functionality

After discovering the MeshAccess Sservice and registering notifications for the TX characteristic, the ResolverConnection will instantiate a MeshAccessConnection. The MeshAccessConnection waits for an encryption handshake within a few seconds. If the handshake is not completed within this time, it will disconnect the connection to save resources.

The maximum transmission unit (MTU) of the MeshAccessConnection is fixed at 16 bytes of payload together with 4 bytes if message integrity code (MIC) which gives a total of 4 bytes.

Exemplary Encryption Handshake

This section shows an exemplary encryption handshake, which should make it easier to implement a smartphone app that supports the MeshAccessConnection protocol.

Encrypt Custom Start

Once a Central is connected to a Peripheral, the MeshAccess Service was discoverd and Notifications were set up properly, the Central is expected to start the Encryption Handshake by sending the ENCRYPT_CUSTOM_START packet.

The packet structure is also documented as part of the MeshAccessModule.

This packet contains:

- senderId (or NODE_ID_APP_BASE (32000) in case the Central is a Smartphone)
- the receiverId set to the partners nodeId (or 0 if unknown at this point)
- a version that identifies the type of encryption handshake (currently 1)
- the keyId, which specifies if e.g. NODE_KEY, NETWORK_KEY, etc... should be used for the connection. Different keys have different permissions and only certain messages can be sent depending on the key.
- the tunnelType which specifies if this is a peer to peer connection or if one side of the connection has other nodes attached that will take part in the communication.

The following is an examplary ENCRYPT_CUSTOM_START packet sent from a Central to a Periphal which is sent in cleartext:

```
19:01:00:00:00:01:02:00:00:00:00
```

Data	Description

Data	Description
0x19	MessageType::ENCRYPT_CUSTOM_START (25)
0x0001	Sent from nodeId 1
0x0000	Sent to NODE_ID_BROADCAST, but will only be received by our connection partner during the handshake
0x01	Encryption Protocol version is currently 1
0x00000002	KEY_ID_NETWORK should be used
0x00	MESH_ACCESS_TUNNEL_TYPE_PEER_TO_PEER is used

Encrypt Custom ANonce

Once the peripheral receives the ENCRYPT_CUSTOM_START packet and wants to accept the connection, it must respond with the ENCRYPT_CUSTOM_ANONCE packet. This packet is also sent in cleartext. To generate this packet it has to use a secure random number generator to generate an 8 byte integer called ANonce:

```
1A:02:00:01:00:1D:4C:FA:4E:32:19:68:2A
```

Data	Description
0x1A	MessageType::ENCRYPT_CUSTOM_ANONCE (26)
0x0002	SenderId is 2
0x0001	ReceiverId is 1
0x4EFA4C1D	First part of the randomly generated ANonce
0x2A681932	Second part of the ANonce

The Peripheral will use the ANonce to decrypt all packets that it receives from the central from now on. To do this, it has to generate the Session Key for decrypting packets. The NetworkKey that was chosen for this example is: `04:00:00:00:00:00:00:00:00:00:00:00:00:00:00:00` . This key must be known by both devices in order to successfully encrypt the connection. The used key is called Long Term Key and is used to generate a Session Key that is only valid for the current connection.

To generate the Session Key, the Peripheral has to create the following cleartext first:

```
01:00:1D:4C:FA:4E:32:19:68:2A:00:00:00:00:00:00
```

Data	Description
0x0001	Node Id of the Central is 1
0x4EFA4C1D	First part of the randomly generated ANonce
0x2A681932	Second part of the ANonce
0x000000000000	Padded with zeros

This cleartext is then encrypted using AES-128 in Electronic Code Book Mode (ECB), which basically means that AES-128 is used without any chaining. If an IV is requested, set it to 00:00:00:00:00:00:00:00:00:00:00:00:00:00:00:00. The result of the encryption is the Session Decryption Key for the Peripheral, which is `03:1C:BD:BA:73:42:FD:B0:95:13:81:AB:97:94:8C:D9` in this example.

Once the packet is received by the Central, it will save the received nonce and will generate the same Session Key so that it can use this Session Key for encrypting packets. As it has received the ANonce in cleartext and as it knows the Long Term Key, it can follow the same steps to derive the exact same Session Key.

Encrypt Custom SNonce

The Central must now also generate a unique 8 byte random number that is called SNonce. It must then send a packet to the Peripheral that contains the SNonce. This packet must be sent in its encrypted form by using the Session Key that was generated using the ANonce.

The cleartext of the ENCRYPT_CUSTOM_SNONCE packet might look like this:

`1B:01:00:02:00:FC:D3:B8:64:AD:0F:E8:19:00:00:00`

Data	Description
0x1B	MessageType::ENCRYPT_CUSTOM_SNONCE (27)
0x0001	Sent from nodeId 1
0x0002	Sent to nodeId 2
0x64B8D3FC	First part of the randomly generated SNonce
0x19E80FAD	Second part of the SNonce
0x000000	Padded with zeros to have a block of 16 bytes

The Central has already calculated the encryption key from the ANonce as described above, it can now also generate the Session Key for decrypting packets by using the generated SNonce. Using the same procedure as above, it has to first construct the cleartext and then encrypt this cleartext by using the Long Term Key.

`01:00:FC:D3:B8:64:AD:0F:E8:19:00:00:00:00:00:00`

Data	Description
0x0001	Node Id of the Central is 1
0x64B8D3FC	First part of the randomly generated SNonce
0x19E80FAD	Second part of the SNonce
0x000000000000	Padded with zeros

After encrypting this, it will have derived the Session Key for decrypting messages from the Peripheral which is `A4:13:1A:68:D2:64:B6:55:90:6E:87:AD:5F:BF:F0:A0` in this example.

Generating the Keystream and Encrypting the Data

Next, it must encrypt the ENCRYPT_CUSTOM_SNONCE packet before it can be sent to the Peripheral by using the Session Encryption Key. To do this, it will first generate a cleartext from the ANonce, encrypt this with the Session Encryption key and then XOR this keystream with the cleartext data to be sent.

In this example, the cleartext to generate the keystream is `1D:4C:FA:4E:32:19:68:2A:00:00:00:00:00:00:00:00`.

Data	Description
0x4EFA4C1D	First part of the ANonce
0x2A681932	Second part of the ANonce
0x0000000000000000	Padded with zeros

Encrypting this cleartext with the Session Encryption Key (`03:1C:BD:BA:73:42:FD:B0:95:13:81:AB:97:94:8C:D9`) results in a keystream that is `62:64:A5:B4:A6:5B:8B:31:69:45:78:05:C5:26:69:E4`. After this keystream was XOR'ed with the data that should be sent (`1B:01:00:D1:07:FC:D3:B8:64:AD:0F:E8:19:00:00:00`), we get the ciphertext, which is `79:65:A5:B6:A6:A7:58:89:0D:E8:77:ED:DC:26:69:E4`. As the keystream was only XOOed with the original data, we can omit the last bytes that were only padded with zeros and only need to send the part that contains data, which is `79:65:A5:B6:A6:A7:58:89:0D:E8:77:ED:DC`.

Generating the MIC

To be able to check, whether the packet was properly encrypted, a message integrity code (MIC) needs to be attached to the packet. Calculating the MIC is done by increasing the encryption nonce by 1 to generate a new cleartext. The new cleartext in this example is `1D:4C:FA:4E:33:19:68:2A:00:00:00:00:00:00:00:00`. Notice the 0x33 which is where the second part of the ANonce was increased by 1. This cleartext is then encrypted with our Session Encryption Key to get the MIC keystream, which is `04:14:36:4B:49:93:9C:40:68:49:7A:55:73:AB:E7:73`.

To finally generate the MIC, the MIC keystream is again XOR'ed with the original cleartext data packet in its zero padded form (`79:65:A5:B6:A6:A7:58:89:0D:E8:77:ED:DC:00:00:00`) which results in `7D:71:93:FD:EF:34:C4:C9:65:A1:0D:B8:AF:AB:E7:73` . Once this is encrypted again with the Session Encryption Key, we have the final data for the MIC `CA:CA:47:57:CB:48:41:6D:56:12:F5:63:DE:10:2C:D3` from which we use the first 4 bytes (CA:CA:47:57) and append it to the end of our encrypted data packet:

```
79:65:A5:B6:A6:A7:58:89:0D:E8:77:ED:DC:CA:CA:47:57
```

This data is then sent to the Peripheral where it can be decrypted using the same procedure as above as XOR'ing data twice will give the initial data.

Decrypting the Packet on the Peripheral

The Peripheral will follow the same procedure that was also used during encryption once it receives the encrypted data packet (`79:65:A5:B6:A6:A7:58:89:0D:E8:77:ED:DC:CA:CA:47:57`).

First, it must check if the MIC matches and must only decrypt the packet if this is the case. Therefore it will first generate a cleartext with the ANonce that it generated initially incremented by one (`1D:4C:FA:4E:33:19:68:2A:00:00:00:00:00:00:00:00`) which it will encrypt with its Session Decryption Key (`03:1C:BD:BA:73:42:FD:B0:95:13:81:AB:97:94:8C:D9`) which produces the MIC Keystream `04:14:36:4B:49:93:9C:40:68:49:7A:55:73:AB:E7:73` . This is now XOR'ed with the zero padded ciphertext (aka. the encrypted data packet: `79:65:A5:B6:A6:A7:58:89:0D:E8:77:ED:DC:00:00:00`) to get `7D:71:93:FD:EF:34:C4:C9:65:A1:0D:B8:AF:AB:E7:73` . After encrypting this with the Session Decryption Key it will have produced the exact same MIC keystream `CA:CA:47:57:CB:48:41:6D:56:12:F5:63:DE:10:2C:D3` and can check if the first 4 bytes match the received MIC.

After checking the MIC, the Peripheral is ready to decrypt the data for which it has to use the original ANonce (Decremented by 1 compared with the ANonce that was used to calculate the MIC) which is `1D:4C:FA:4E:32:19:68:2A:00:00:00:00:00:00:00:00` . Encrypting this with the Session Decryption Key results in `62:64:A5:B4:A6:5B:8B:31:69:45:78:05:C5:26:69:E4` . After this was XOR'ed with the encrypted data packet, we get the original SNonce data packet:

```
1B:01:00:D1:07:FC:D3:B8:64:AD:0F:E8:19:00:00:00
```

As the length of this packet is known, we can now remove the zero padding. The Peripheral must then store the received SNonce and use it to encrypt all further packets

Encrypt Custom Done

The Peripheral can now use the encryption nonce and can generate the Session Encryption Key before it sends the final ENCRYPT_CUSTOM_DONE handshake packet:

```
1C:02:00:01:00:00
```

Data	Description
0x1C	MessageType::ENCRYPT_CUSTOM_DONE (28)
0x0002	SenderId
0x0001	ReceiverId

Data	Description
0x00	Status OK

This data is now encrypted again in the same manner as before:

- Session Encryption Key: `A4:13:1A:68:D2:64:B6:55:90:6E:87:AD:5F:BF:F0:A0`
- Keystream Cleartext from SNonce: `FC:D3:B8:64:AD:0F:E8:19:00:00:00:00:00:00:00:00`
- Data Encryption Keystream: `83:30:E5:B0:4F:7B:EB:04:92:D8:75:84:DC:80:54:FE`
- Data Cleartext: `1C:02:00:01:00:00:E8:19:00:00:00:00:00:00:00:00`
- Data Cleartext XOR'ed with Encryption Keystream (Data Ciphertext):
  `9F:32:E5:B1:4F:7B:03:1D:92:D8:75:84:DC:80:54:FE`

MIC calculation:

- MIC Intermediate Keystream Cleartext: `FC:D3:B8:64:AE:0F:E8:19:00:00:00:00:00:00:00:00`
- MIC Intermediate Keystream: `0B:B1:CC:00:F4:53:33:92:3B:A7:71:98:FF:77:95:4E`
- Padded Data Ciphertext: `9F:32:E5:B1:4F:7B:00:00:00:00:00:00:00:00:00:00`
- Padded Data Ciphertext XOR'ed with MIC Intermediate Keystream:
  `94:83:29:B1:BB:28:33:92:3B:A7:71:98:FF:77:95:4E`
- Above thingy encrypted with Session Encryption Key (MIC Data):
  `62:92:E7:B6:4C:A0:88:E2:B0:7E:95:87:01:84:01:86`
- MIC: `62:92:E7:B6`

Final Encrypted Packet:

- `9F:32:E5:B1:4F:7B:62:92:E7:B6`

Final Words

Hopefully, the above examples together with the source code in the MeshAccessConnection should provide enough help to be able to implement the complete Handshake and the following encryption of all data packets. Keep in mind that the ANonce and SNonce are increased by two for every data packet as one Nonce is used to encrypt the data and the second Nonce is used to generate the MIC.

Data splitting happens before encryption, so if a message is sent or received that has more than 16 bytes, it must be split into multiple parts before the individual parts are encrypted.