Contents

[Intro 1](#_Toc1399038)

[1. CanPlugs auto provisioning model 2](#_Toc1399039)

[2. HSM 4](#_Toc1399040)

[3. Attestation mechanisms 5](#_Toc1399041)

[3.1 X.509 - Single enrollment 5](#_Toc1399042)

[3.2 X.509 - Group enrollment 6](#_Toc1399043)

[3.3 Symmetric Key - Single enrollment 7](#_Toc1399044)

[3.4 Symmetric Key - Group enrollment 8](#_Toc1399045)

# Intro

Before reading this document please get familiar with Concepts of DPS: <https://docs.microsoft.com/en-us/azure/iot-dps/concepts-auto-provisioning>

Acronyms used in this document:

Azure Op – Azure Operator

Factory Op – Factory Operator

TPM – Trusted Platform Module

HSM – Hardware Security Module

DPS – Device Provisioning Service

EL – Enrollment List

DDK – Derived Device Key

# CanPlugs auto provisioning model

1. Azure DPS. Deployed and linked with single or multiple Azure IoT Hubs. EL for single or group provisioning created manually by Azure Op or with help of provisioning tools
2. Provisor. Small tool that should start during first device boot, connect to DPS and use selected attestation method to provision device in Azure IoT Hub. This functionality can be incorporated into exiting Azure IoT Hub Client application if:  
   a) Factory is going to deploy final CanPlug software during production  
   b) CanPlugs should be designed for re-provisioning
3. HSM – adapter for Azure IoT Provisioning SDK. Holds device identity and authentication information. Can be implemented as:  
   a) Software with file system based storage

b) Software with embedded into binary storage

c) Software that stands as bridge between firmware storage and consumer

1. Provisioning tools and scripts. Helper tools to generate symmetric keys and DDK, create chain of certificates, automate creation of EL in DPS, etc.
2. Azure IoT Hub client. Modified version of existing CanPlug software that uses custom HSM to authenticate device in Azure IoT Hub

There are 3 device attestation mechanisms in DPS:

* TPM
* X.509
* Symmetric Key

Some of them support Group Enrollment, and this case is threated as separate attestation type in this document. TPM attestation is omitted as option of auto-provisioning for CanPlugs because of the following reasons:

* There are no hardware/OS support of TPM 2.0 standard in Collibri IMX6 and Angstrom Linux
* Implementing Firmware TPM or Software emulated TPM brings a lot of difficulty, since requires integration of 3rd party cryptographic library. Moreover, additional unit testing is required to ensure this TPM implementation meets standard.
* Firmware/Software TPM is a compromise to Discrete/Integrated TPMs in any case, because EK and SRK values can be discovered by malefactor sooner or later.

Please find additional information regarding TPM attestation below:

<https://docs.microsoft.com/en-us/azure/iot-dps/concepts-tpm-attestation>

**Popular types of TPM**

* **DISCRETE TPM (TPM 1.2 & TPM 2.0)**  
  *Discrete TPM provides the highest level of security. The intent of this level is to ensure that the device it’s protecting does not get hacked via even sophisticated methods. To accomplish this, a discrete chip is designed, built and evaluated for the highest level of security that can resist tampering with the chip, including probing it and freezing it with all sorts of sophisticated attacks.*
* **INTEGRATED TPM**  
  *Integrated TPM is the next level down in terms of security. This level still has a hardware TPM but it is integrated into a chip that provides functions other than security. The hardware implementation makes it resistant to software bugs, however, this level is not designed to be tamper-resistant.*
* **FIRMWARE TPM (fTPM)**  
  *Firmware TPM is implemented in protected software. The code runs on the main CPU, so a separate chip is not required. While running like any other program, the code is in a protected execution environment called a trusted execution environment (TEE) that is separated from the rest of the programs that are running on the CPU. By doing this, secrets like private keys that might be needed by the TPM but should not be accessed by others can be kept in the TEE creating a more difficult path for hackers. In addition to the lack of tamper resistance, the downside to the TEE or firmware TPM is that now the TPM is dependent on many additional aspects to keep it secure, including the TEE operating system, bugs in the application code running in the TEE, etc.*
* **SOFTWARE TPM**  
  *Software TPM can be implemented as a software emulator of the TPM. However, a software TPM is open to many vulnerabilities, not only tampering but also the bugs in any operating system running it. It does have key applications: it is very good for testing or building a system prototype with a TPM in it. For testing purposes, a software TPM could provide the right solution/approach.*

# HSM

For other types of attestation a custom implementation of HSM is required. Main responsibility of this module is to keep secret information in relative safety. And provide this info on demand to Azure Provisioning/Client SDK for attestation in DPS and further connection to Azure IoT Hub. List of information that HSM should provide depends on selected attestation mechanism:

1. Registration ID: Unique identifier of device in the DPS. May be the same as final DeviceID in Azure IoT Hub, or some hardware related info, like MAC, IMEI, PCB serial No
2. Symmetric Key: Primary Key in case of individual enrollment or DDK in case of group enrollment
3. X.509 Private/Public keys: Leaf X.509 certificate, derived from Root CA or intermediate certificate. Public part is for authentication and Private for proof-of-possession

Depending on desired level of security this module can be implemented as:

a) Software with file system based storage: static library that provides read-only “secret” information from file system. Most simple and less secure variant among others. Advantages include ease of deployment – certificates and symmetric key can be copied easily by scripts to destination CanPlug

b) Software with embedded into binary storage. Secrets are injected into module binary code during compilation. Requires automation script to build individual binary for each CanPlug. Adds some complexity to decode and extract secrets.

c) Software that stands as bridge between firmware storage and consumer. Real secrets are kept in the microcontroller flash memory, while firmware provides it to HSM over serial port and API. Most secure way among other, but adds a lot of complexity to implementation/deployment:

- Need to develop new version of firmware and API

- HSM implementation becomes more complex, since includes communication with microcontroller

- Deployment process on factory now should cover both: firmware/software components of CanPlug

# 3. Attestation mechanisms

## 3.1 X.509 - Single enrollment

Scenario:

* + 1. Azure Op creates self-signed or order authorized Root CA. Installs this certificate into DPS
    2. Azure Op, deriving from Root CA, generates list of device certificates using script
    3. Azure Op sends this list to the Factory Op
    4. Azure Op, using provisioning tool, creates EL in DPS from list of device certificates. Enrollment policy and initial Device Twin should be specified at this stage
    5. Factory Op uses entries in provided list to inject provisioning software and device certificate during production of CanPlugs

Device Input: ScopeID, X.509 PK/Pub, Registration ID

Factory Input: ScopeID, List of ( X.509 PK/Pub + Registration ID)

HSM implementation complexity: **low**

Provisioning tool complexity: **medium**

Provisioning process complexity: **medium**

Provisioning flexibility: **low**

Security Vulnerabilities: only single device in IoT hub is affected in case of compromised CanPlug hardware

Pros: ease of implementation

Cons: Need to synchronize EL between factory and operator. Additional effort needed to manage provisioning policy for each device. Additional script to generate device certificate

## 3.2 X.509 - Group enrollment

Scenario:

1. Azure Op creates self-signed or order authorized Root CA. Installs this certificate into DPS
2. Azure Op, deriving from Root CA, creates Factory Certificate
3. Azure Op sends this single certificate to Factory Op
4. Based on mutual with Azure Op agreement, Factory Op generates chain of certificates. For instance per model, version, groups of canplugs, production month, etc. List of sub-certificates is sent back to Azure Op

4a) As alternative approach, chain of sub-certificates can be created by Azure Op and sent to Factory Op

1. Azure Op creates Enrollment Group Entries in DPS using sub-certificates. Enrollment policy and Device Twin is common for whole Enrollment Group
2. Factory Op generates Leaf certificates as children of previously created sub-certificates
3. Factory Op injects provisioning software and device certificate during production of CanPlugs

Device Input: ScopeID, X.509 PK/Pub, Registration ID

Factory Input: ScopeID, Intermediate certificate(s)

HSM implementation complexity: **low**

Provisioning tool complexity: **medium**

Provisioning process complexity: **high**

Provisioning flexibility: **high**

Security Vulnerabilities: only single device can be compromised. Possible leakage of intermediate certificate from factory, leading to possible cloning of device certificates

Pros: ease of implementation, selectivity in the enrollment policies, ability to manage group of devices with ease, or override policy for individual device

Cons: complexity in certificate chain management

## 3.3 Symmetric Key - Single enrollment

Scenario:

1. Azure Op uses script to generate list of pairs: Symmetric Key and Registration ID
2. Azure Op uses provisioning tool to create EL in DPS using list created in previous step
3. Azure Op sends list to Factory Op
4. Factory Op uses provided list to inject provisioning software and SymKey+Registration ID into CanPlug during production

Device Input: ScopeID, Symmetric Key, RegistrationID

Factory Input: ScopeID, EL

HSM implementation complexity: **low**

Provisioning tool complexity: **low**

Provisioning process complexity: **low**

Provisioning flexibility: **low**

Security Vulnerabilities: risk of single device faking

Pros: ease of implementation

Cons: Need to synchronize EL between factory and operator. Additional effort needed to manage provisioning policy for each device. No options to manage group of devices

## 3.4 Symmetric Key - Group enrollment

Scenario:

1. Azure Op uses script to create Master Key(s)
2. Azure Op sends Master Key(s) to Factory Op and agrees how those keys should be devoted
3. Azure Op prepares Enrollment Groups in DPS using Master Key(s)
4. Factory Op uses script to prepare list of DDKs using Master Key(s)
5. Factory Op uses list of DDKs to inject provisioning software and DDK+Registration ID into CanPlug during production

Device Input: ScopeID, DDK, RegistrationID

Factory Input: ScopeID, Master Symmetric Key(s)

HSM implementation complexity: **low**

Provisioning tool complexity: **medium**

Provisioning process complexity: **medium**

Provisioning flexibility: **medium**

Security Vulnerabilities: risk of single device faking. Risk of device spawning in case of Master Symmetric Key leakage

Pros: ease of implementation, simple enrollment policy management

Cons: additional script required for factory to create DDKs