AN13500

EdgeLock A5000 Secure Authenticator for electronic anti-counterfeit protection using device-to-device authentication

Rev. 1.1 — 14 September 2022 Application note

Document information

Information	Content
Keywords	A5000, mutual authentication, proof of possession
Abstract	This document describes how to leverage A5000 for device-to-device authentication

Revision history

Revision history

Revision number	Date	Description
1.0	2022-03-28	Initial version
1.1	2022-09-14	Update Section 4.8.3 How to configure the A5000 product specific SCP keys in the Plug & Trust Middleware.

1 Device-to-device authentication

The IoT environment increases the exposure of high value components to new security threats. OEM manufacturers need to protect themselves from non-authorized components, discriminate original devices from fake copies, avoid device misuse and over usage, and make sure customers purchase original equipment.

If we do not take security into account, attackers may try to compromise our devices by:

- Exploiting software bugs
- Extracting secret device keys
- Inserting counterfeit devices
- Abusing untrusted connections
- Disclosing confidential data, etc

These security threats are significantly serious for IoT systems dealing with real time processes, even risking safety in case of medical devices, industrial processes, energy grids or traffic lights automation, among others.

For illustrative purposes, let's assume an OEM which manufactures a certain type of machinery controlled by a centralized control unit as shown in Figure 1. As these machines perform some critical tasks in the manufacturing plant:

- The control unit authenticates the machine that is attempting to connect to it.
- The machines also authenticate the control unit that will manage it.

Therefore, only authenticated machines and control units will be used in the supply chain. This mechanism ensures protection against rogue devices that might damage production, degrading security levels or risking employee safety.

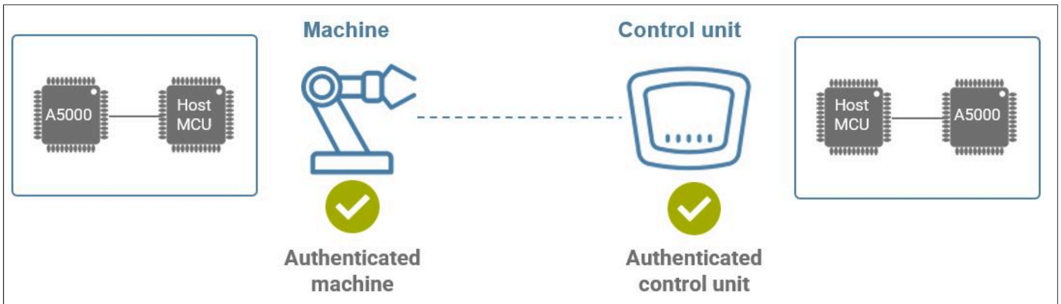


Figure 1. Device-to-device authentication scenario

The exchange of digital certificates is the basis of the authentication process. The two parties each check that the certificate is valid and was issued by a trusted authority, known as Certificate Authority. Section 2 describes how certificates are verified using a certificate chain of trust.

Digital certificates, as public information, are susceptible to be intercepted and be misused. For this reason, a proof of possession of the certificate private key is an essential requirement to validate the certificate source. Section 3 describes how to leverage A5000 to conduct the proof of possession.

The private key must be kept secret and protected. The leakage of any private key compromises the identity verification and the overall system security. The A5000 provides a trust anchor at the silicon level, providing a tamper-resistant platform capable of securely storing keys and credentials needed for offline authentication.

2 Certificate chain of trust

IoT requires each device to possess a unique identity. For certificate-based authentication scheme, the identity is made of:

- Device certificate
- Device key pair

The digital certificate binds an identity with a public key. Digital certificates are verified using a chain of trust. The certificate chain of trust is a structure of certificates that enable the receiver to verity that the sender and all CA's are trustworthy. The trust anchor for the digital certificate is the root CA.

Certificates are issued and signed by certificates that reside higher in the certificate hierarchy, so the validity and trustworthiness of a given certificate is determined by the corresponding validity of the certificate that signed it. The certificate chain of trust results in a root CA signing an intermediate CA that in turn signs a leaf certificate as shown in Figure 2

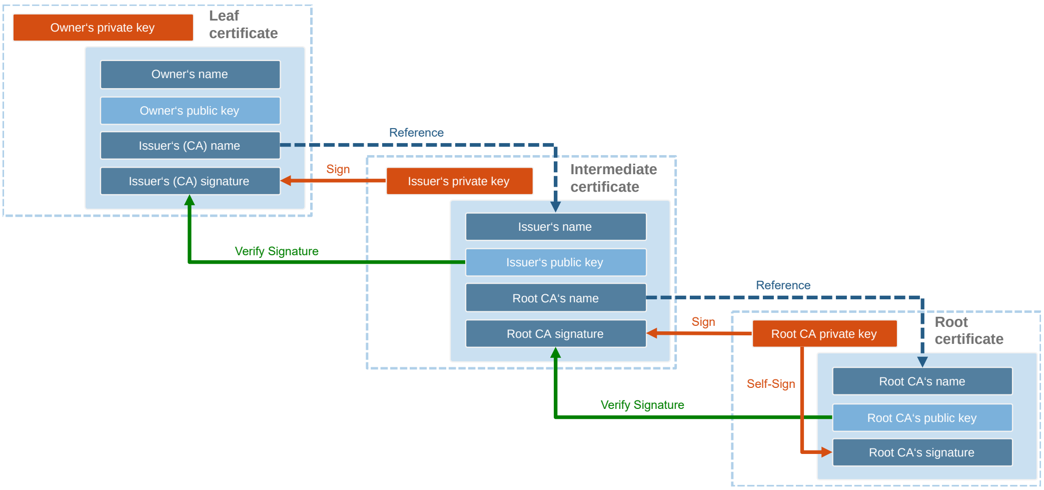


Figure 2. Certificate chain of trust

IoT devices manufactured by the OEM should be equipped with a unique key pair and a digital certificate signed by the OEM's CA certificate. The OEM's CA certificate is used to sign all the certificates of the devices manufactured by the OEM. Precisely, this signature provides the means to verify the validity of device certificates in the field (Figure 3).

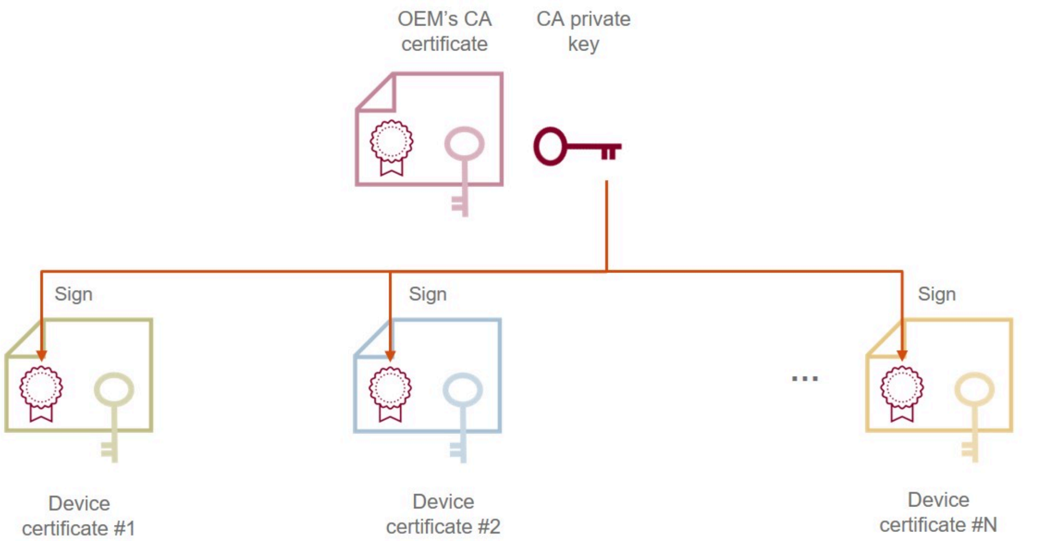


Figure 3. Certificate hierarchy

Before a machine or control unit manufactured by the OEM goes to the operation phase, they must possess the CA certificate, an individual certificate and a key pair securely stored as shown in Figure 4.

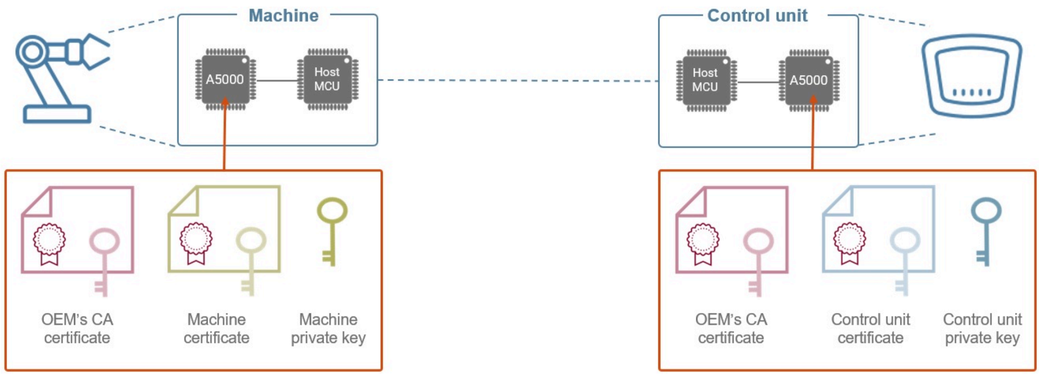


Figure 4. Machine and control unit credentials

Secure silicon chips like A5000 are capable of internally protecting private keys in IoT devices. The CA certificate could optionally be stored outside the A5000. Section 5 outlines the A5000 trust provisioning models available.

3 Mutual authentication flow

The authentication flow consists of a mutual authentication procedure. First, the machine will authenticate the control unit that it will be connected to. After that, the control unit will authenticate the machine that attempts to connect.

3.1 Control unit authentication

The authentication of the control unit consists of two steps: the *certificate validation* and the private key *proof of possession* as shown in Figure 5.

Certificate validation:

The first step is the verification of the control unit digital certificate.

1. The control unit sends its device certificate together with its hierarchy of CA certificates.
2. The machine validates that the provided certificate chain of trust is valid by verifying the signatures of all the certificates in the chain up to the root CA

If the control unit certificate is valid, it means that the public key included in it can be trusted.

Proof of possession:

The second step is the proof of possession. This procedure is needed to make sure that the certificate we verified belongs to the control unit. This proof of possession mechanism ensures that the uploader of the certificate also knows the associated private key. For that,

1. The machine generates a random challenge
2. The control unit returns the random challenge signed, using its private key stored inside A5000.
3. The machine validates the random number signature with the public key obtained from the control unit certificate.

A successful response means that the control unit is authentic. Bear in mind that the trust relies on protecting the private key. For this reason, the use of A5000 is fundamental to make sure the private key is not compromised.

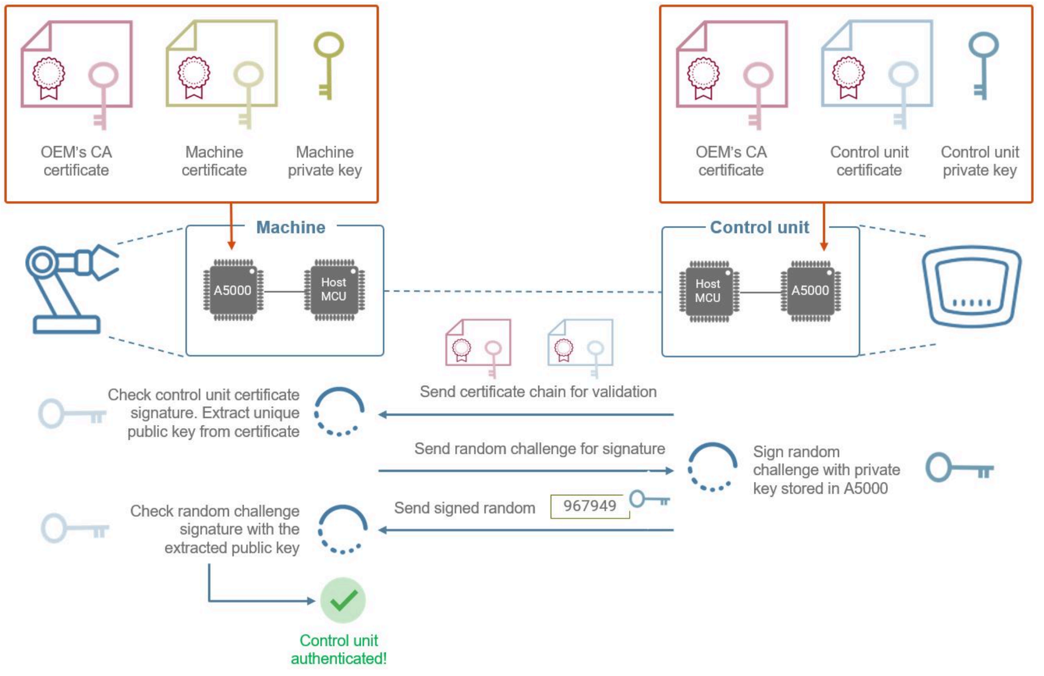


Figure 5. Control unit authentication flow

3.2 Machine authentication

The authentication of the machine also consists of two steps: the *certificate validation* and the private key *proof of possession* as shown in Figure 6. These two steps are equivalent to the ones performed for the control unit authentication.

Certificate validation:

The first step is the verification of the machine digital certificate.

1. The machine sends its device certificate together with its hierarchy of CA certificates.
2. The control unit validates that the provided certificate chain of trust is valid by verifying the signatures of all the certificates in the chain up to the root CA

If the machine certificate is valid, it means that the public key included in it can be trusted.

Proof of possesion:

The second step is the proof of possession. This procedure is needed to make sure that the certificate we received belongs to the machine. This proof of possession mechanism ensures that the uploader of the certificate also knows the associated private key. For that,

1. The control unit generates a random challenge
2. The machine returns the random challenge signed, using its private key stored inside A5000.
3. The control unit validates the random number signature with the public key obtained from the machine certificate

A successful response means that the machine is authentic. Bear in mind that the trust relies on protecting the private key. For this reason, the use of A5000 is fundamental to make sure the private key is not compromised.

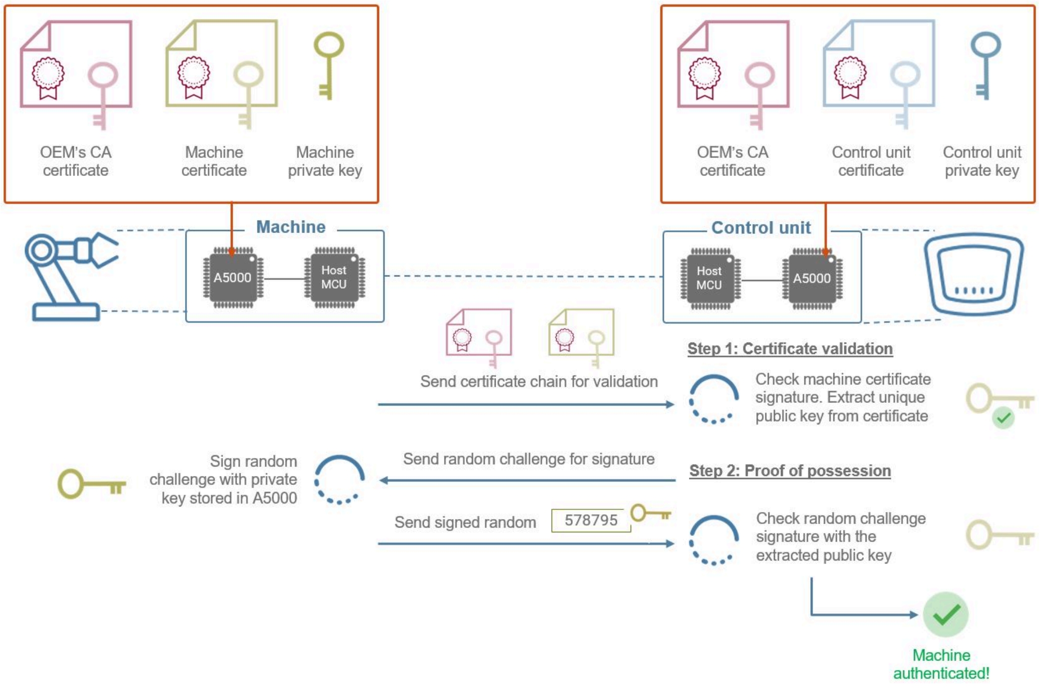


Figure 6. Machine authentication flow

4 Evaluating A5000 for anticounterfeit protection

This chapter describes how to evaluate the A5000 Secure Authenticator for anticounterfeit protection using device-to-device authentication. The following description is provided only for demonstration. Therefore, the subsequent procedure must be adapted and adjusted accordingly for commercial deployment.

The Plug Trust Middleware offers out of the box several software libraries to implement and verify a device-to-device authentication on devices running an embedded Linux distribution.

- OpenSSL
- PKCS11
- Plug&Trust Middleware SSS API

The following chapters are demonstrating the principal of the machine and control unit authentication flow based on the theoretical example described in Section 3". To simplify the hardware setup a single A5000 Secure Authenticator IC is used.

To keep the example as simple as possible only A5000 pre-provisioned credentials are used to demonstrate the Mutual Authentication.

The examples are divided into the following steps to introduce the A5000 and the Plug&Trust OpenSSL engine and the ssscli tools:

1. Section 4.1 Hard- and software setup
2. Section 4.2 OpenSLL engine overview
3. Section 4.3 Plug & Trust Middleware ssscli tool introduction
4. Section 4.4 Pre-provisioned A5000 device certificates used by the example
5. Section 4.5 Retrieve the pre-provisioned A5000 credentials
6. Section 4.6 Chain of trust of the pre-provisioned device certificates
7. Section 4.7 Mutual authentication flow
   a. Section 4.7.1 Control unit authentication
   b. Section 4.7.2 Machine authentication

The physical I2C connection between the Raspberry Pi and the A5000 Secure Authenticator can be established either in plain or secured (authenticated and encrypted) using the Global Platform Secure Channel Protocol 03 (SCP03). Section 4.8 gives a brief overview about Global Platform Secure Channel Protocol 03 and explains how to run the examples using Platform SCP.

How to manage access from multiple Linux processes to the A5000 authenticator application is briefly discussed in Section 4.9 .

4.1 Hard- and software setup

The following hardware is used for this demo as a reference for any other embedded Linux board like the NXP i.MX8:

- Raspberry Pi3 Model B+ or Pi4 Model B
- OM-A5000ARD development kit (NXP 12NC: 935424319598)
- Optional - OM-SE050RPI adapter board for Raspberry Pi (NXP 12NC: 935379833598)

The AN12570 "Quick start guide with Raspberry Pi" describes the hardware and software for the NXP SE05x Secure Element. Chapter "3.3. Build EdgeLock SE Plug & Trust

Middleware test examples” describes the CMake settings to build the middleware accessing a SE05x Secure Element.

To build the Plug & Trust Middleware to support the A5000 Secure Authenticator application the following CMake setting needs to be modified before building the middlware:

- Select ***AUTH*** *for the CMake option* ***PTWM_Applet***.
- Select ***None*** for the CMake option ***PTWM_FIPS***.
- Select ***07_02*** *for the CMake option* ***PTWM_SE05X_Ver***.
- Disable the CMake option **SSSFTR_SE05X_RSA**.

The project settings can be specified dynamically using the CMake GUI. Figure 7 shows a CMake GUI screenshot with EdgeLock A5000 project settings.

- Run the following commands to update the CMake settings and rebuild the Plug & Trust Middleware:
  ```
  cd ~/se_mw/simw-top_build/raspbian_native_se050_t1oi2c
  cmake-gui .
  ```
  Update the CMake settings as explained above. Press first ***Configure*** and second ***Generate*** and close the CMake GUI.
  ```
  cmake --build .
  sudo make install
  sudo ldconfig /usr/local/lib/
  ```

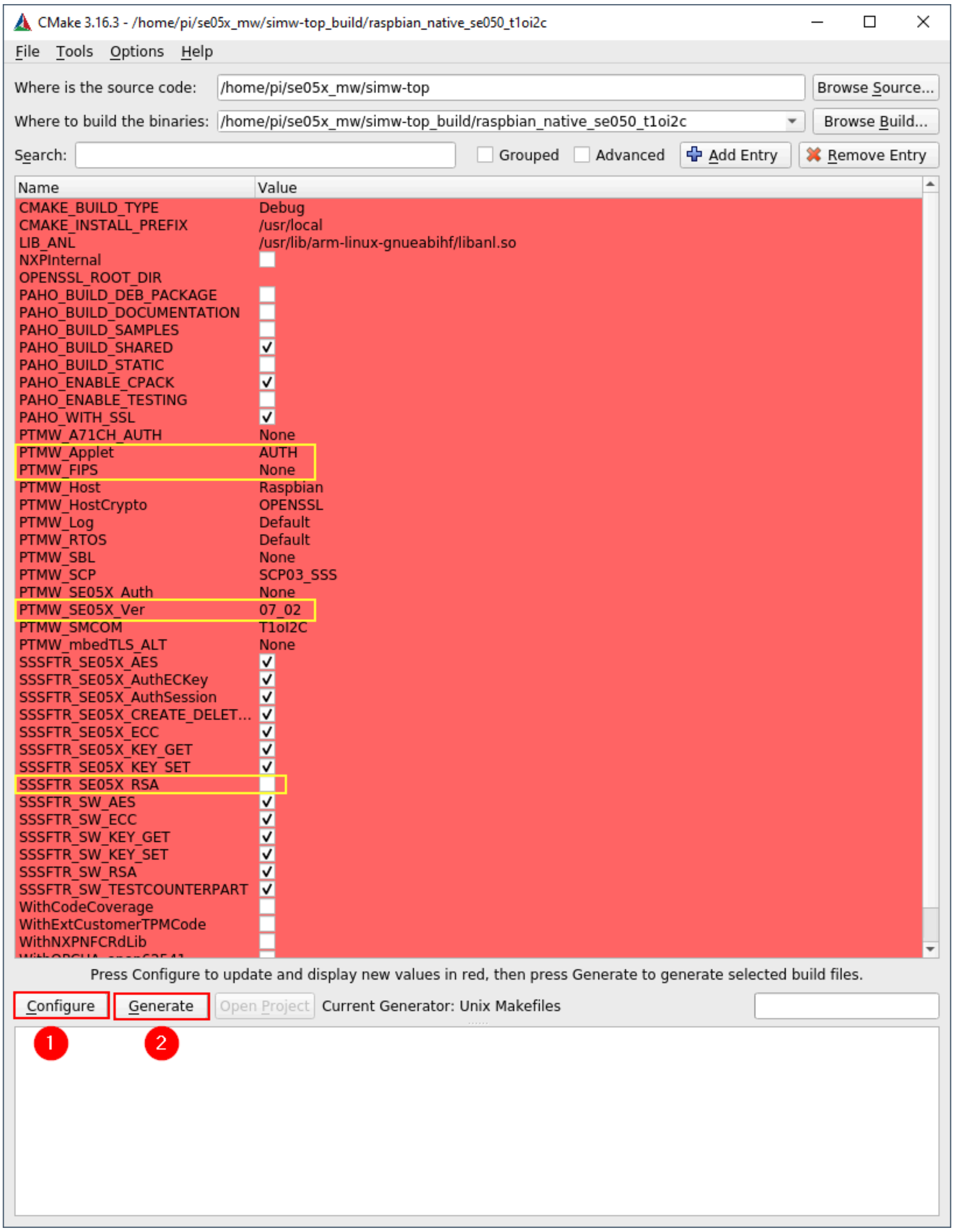


Figure 7. A5000 CMake options

4.2 OpenSSL engine overview

OpenSSL is a free software library contains an open-source implementation of the TLS protocols. OpenSSL is available for most Unix-like operating systems (including Linux, macOS, and BSD) and Microsoft Windows.

The OpenSSL software library, written in C, includes a command-line interface for general-purpose cryptography and managing certificates. For simplification the demos below are using the OpenSSL CLI.

Starting with OpenSSL version 0.9.6, a new component called ENGINE, was added to support alternative cryptography implementations. This Engine interface is used by the Plug & Trust Middleware to interface with the A5000. The OpenSSL engine provides the glue between applications using standard OpenSSL APIs and the Secure Authenticator API.

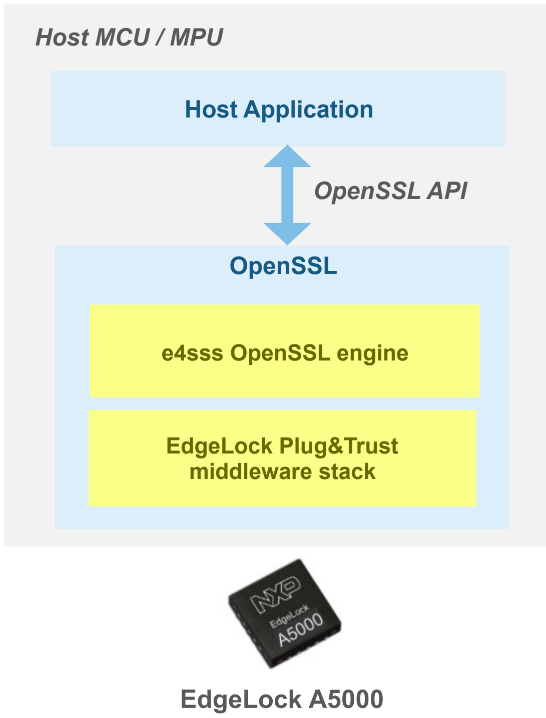


Figure 8. Principle of the OpenSSL engine

The Plug&Trust middleware OpenSSL engine allows to use the A5000 Secure Authenticator for the following operations:

- EC crypto: EC sign/verify and ECDH compute key
- Fetching random data

The A5000 secure key and object management is not covered by the engine interface but supported by the Plug & Trust Middleware `ssscli tool` as demonstrated in the next chapters.

OpenSSL requires a key pair, consisting of a private and a public key, to be generated or loaded into the A5000 before the cryptographic operations can be executed.

- **Private Key**: The Private key is securely stored inside the A5000 Secure Authenticator and cannot be retrieved by the OpenSSL engine.
- **Reference Key**: Standard OpenSSL API needs to be called with a key. Instead of a real private key the OpenSSL key data structure gets used with a reference to the private key inside the Secure Authenticator. The reference key looks for OpenSSL like a real key, but it does not contain secret data.

- **Certificate/Public Key**: The Certificate/Public Key as read from the Secure Element can still be inserted into the OpenSSL key structure.

The A5000 Secure Authenticator can be easily integrated by applications which are already using the OpensSSL API or the command-line tools. Instead of using a private key, the application needs to use a reference key.

For more details, please see the Plug & Trust Middleware documentation:

- 8.1. Introduction on OpenSSL engine
- 5.3.2. AWS Demo for iMX Linux / RaspberryPi
- 5.4.2. OpenSSL Engine: TLS Client example for iMX/Rpi3

Run the following command to check weather OpenSSL is installed or not:

```
openssl version
```

Figure 9. Check the installed OpenSSL version

If OpenSSL is not already installed, you can run the following commands to install it:

```
apt-get install openssl libssl-dev
```

4.3 Plug & Trust Middleware ssscli tool introduction

The ssscli is a command line tool that can be used to send commands to A5000 interactively through the command line. For example, you can use the ssscli to create keys and credentials in the A5000 security IC during evaluation, development and testing phases. The ssscli tool is written in Python and supports complex provisioning scripts that can be run in Windows, Linux, OS X and other embedded devices. It can be used to:

- Insert keys and certificates in DER or PEM format into the A5000
- Retrieve the public keys and certificates form A5000 and store the key into a DER (Distinguished Encoding Rules) or PEM (Privacy Enhanced Mail) formatted file
- Create reference-keys and store the key into a DER or PEM formatted file
- Delete A5000 (erase) keys and certificates inside
- Generate keys inside the EdgeLock A5000
- Attach policies to objects
- List all A5000 secure objects
- Retrieve the A5000 device unique ID
- Run some A5000 basic operations like sign/verify and encrypt/decrypt operations

Please refer to the Plug & Trust Middleware documentation chapter "9. CLI Tool" for detailed description how to use ssscli tool. Alternative use the following command to display the ssscli built in help:

```
ssscli --help
```

```
pi@raspberrypi:~ $ ssscli --help
Usage: ssscli [OPTIONS] COMMAND [ARGS]...

  Command line interface for SE050

Options:
  -v, --verbose  Enables verbose mode.
  --version      Show the version and exit.
  --help         Show this message and exit.

Commands:
  a71ch       A71CH specific commands
  cloud       (Not Implemented) Cloud Specific utilities.
  connect     Open Session.
  decrypt     Decrypt Operation
  disconnect  Close session.
  encrypt     Encrypt Operation
  erase       Erase ECC/RSA/AES Keys or Certificate (contents)
  generate    Generate ECC/RSA Key pair
  get         Get ECC/RSA/AES Keys or certificates
  policy      Create/Dump Object Policy
  refpem      Create Reference PEM/DER files (For OpenSSL Engine).
  se05x       SE05X specific commands
  set         Set ECC/RSA/AES Keys or certificates
  sign        Sign Operation
  verify      verify Operation
pi@raspberrypi:~ $
```

Figure 10. ssscli help

The help includes a parameter description for all supported commands. To list all options for the `connect` command use:

```
ssscli connect --help
```

```
pi@raspberrypi:~ $ ssscli connect --help
Usage: ssscli connect [OPTIONS] subsystem method port_name

  Open Session.

  subsystem = Security subsystem is selected to be used. Can be one of
  "se05x, auth, a71ch, mbedtls, openssl"

  method = Connection method to the system. Can be one of "none, sci2c,
  vcom, t1oi2c, jrcpv1, jrcpv2, pcsc"

  port_name = Subsystem specific connection parameters. Example: COM6,
  127.0.0.1:8050. Use "None" where not applicable. e.g. SCI2C/T1oI2C.
  Default i2c port (i2c-1) will be used for port name = "None".

Options:
  --auth_type [None|PlatformSCP|UserID|ECKey|AESKey|UserID_PlatformSCP|ECKey_PlatformSCP|AESKey_PlatformSCP]
                                  Authentication type. Default is "None". Can
                                  be one of "None, UserID, ECKey, AESKey,
                                  PlatformSCP, UserID_PlatformSCP,
                                  ECKey_PlatformSCP, AESKey_PlatformSCP"
  --scpkey TEXT                   File path of the platformscp keys for
                                  platformscp session
  --help                          Show this message and exit.
pi@raspberrypi:~ $
```

Figure 11. ssscli connect help

Note: *The subsystem option* `auth` *shall be used to define a session with the A5000 authenticator. For the Raspberry Pi the connection method* `none` *can be used.*

The A5000 Secure Authenticator supports also the `se05x` commands `certuid`, `readidlist`, `reset` and `uid`.

```
ssscli se05x --help
```

```
pi@raspberrypi:~ $ ssscli se05x --help
Usage: ssscli se05x [OPTIONS] COMMAND [ARGS]...

  SE05X specific commands

Options:
  --help  Show this message and exit.

Commands:
  certuid     Get SE05X Cert Unique ID (10 bytes)
  readidlist  Read contents of SE050
  reset       Reset SE05X
  uid         Get SE05X Unique ID (18 bytes)
pi@raspberrypi:~ $
```

Figure 12. ssscli se05x help

The following commands will list all A5000 secure objects:

```
ssscli connect auth t1oi2c none
```

```
ssscli se05x readidlist
```

```
pi@raspberrypi:~ $ ssscli connect auth t1oi2c none
pi@raspberrypi:~ $ ssscli se05x readidlist
sss   :INFO :atr (Len=35)
      01 A0 00 00    03 96 04 03    E8 00 FE 02    0B 03 E8 00
      01 00 00 00    00 64 13 88    0A 00 65 53    45 30 35 31
      00 00 00
sss   :WARN :Communication channel is Plain.
sss   :WARN :!!!Not recommended for production use.!!!
Key-Id: 0X7da00001   USER-ID
Key-Id: 0X7da00002   AES                          Size(Bits): 128
Key-Id: 0X7da00003   NIST-P        (Public Key)   Size(Bits): 256
Key-Id: 0X7da00011   USER-ID
Key-Id: 0X7da00012   AES                          Size(Bits): 128
Key-Id: 0X7da00013   NIST-P        (Public Key)   Size(Bits): 256
Key-Id: 0X7fff0201   NIST-P        (Key Pair)     Size(Bits): 256
Key-Id: 0X7fff0202   NIST-P        (Key Pair)     Size(Bits): 256
Key-Id: 0X7fff0204   NIST-P        (Public Key)   Size(Bits): 256
Key-Id: 0X7fff0205   USER-ID
Key-Id: 0X7fff0206   BINARY                       Size(Bits): 144    machine
Key-Id: 0X7fff0207   USER-ID                                         credentials
Key-Id: 0X7fff020a   AES                          Size(Bits): 128
Key-Id: 0Xf0000000   NIST-P        (Key Pair)     Size(Bits): 256    ECC256 key pair 0
Key-Id: 0Xf0000001   BINARY                       Size(Bits): 3760   Certificate 0
Key-Id: 0Xf0000002   NIST-P        (Key Pair)     Size(Bits): 256    ECC256 key pair 1
Key-Id: 0Xf0000003   BINARY                       Size(Bits): 3760   Certificate 1
Key-Id: 0Xf0000012   NIST-P        (Key Pair)     Size(Bits): 256    control unit
Key-Id: 0Xf0000020   NIST-P        (Public Key)   Size(Bits): 256    credentials
Key-Id: 0Xf0003394   AES                          Size(Bits): 256

pi@raspberrypi:~ $
```

Figure 13. ssscli readidlist

Note: *If you are not able to connect to the A5000 with an error saying that there is a session already open, run* `ssscli se05x disconnect` *first.*

To close a session use:

```
ssscli disconnect
```

4.4 Pre-provisioned A5000 device certificates used by the example

Examples described in this document are using the following pre-provisioned credentials listed in the table below.

To be able to demonstrate the principle of machine and control unit authentication flow with a single Raspberry Pi and OM-A5000ARD board the ECC256 key pair 0 (object ID `0xF0000000`) and the corresponding certificate 0 (object ID `0xF0000001`) are used as "machine" credentials.

As "control unit" credentials the ECC256 key pair 1 (object ID `0xF0000002`) and the corresponding certificate 1 (object ID `0xF0000003`) are used.

Table 1. Pre-provisioned certificates and keys used by the example

Credentials are assigned to	Key name and type	Certificate	Identifier
Machine	Originality Key 0, ECC256, Die Individual	Certificate 0	0xF0000000 **(key)** 0xF0000001 **(cert)**
Control Unit	Originality Key 1, ECC256, Die Individual	Certificate 1	0xF0000002 **(key)** 0xF0000003 **(cert)**

Note: *The complete list of pre-provisioned device credentials is provided in the A5000 EdgeLock Secure Authenticator Product data sheet.*

4.5 Retrieve the pre-provisioned A5000 credentials

The ECC private keys are securely stored inside the A5000 secure authenticator and cannot be read out. Standard OpenSSL API and the OpenSSL command-line tools needs to be called with a private key to perform private key operations.

The Plug & Trust Middleware provides an OpenSSL engine which allows to use so called reference keys instead of private keys. A reference key contains only a reference, the object ID, to the private key inside the A5000. The reference key looks for OpenSSL like a real key, but it does not contain secret data. All private ECC operations using a reference key, e.g. ECC signing, are performed securely inside the A5000 without the need to know the private key value.

This chapter demonstrates how to use the ssscli and openssl command-line tools to perform the following operations:

- Reading of the pre-provisioned A5000 device certificates and save them into a PEM formatted file.
- Reading of the pre-provisioned A5000 device certificates public keys and save them into a PEM formatted file.
- Creation of the corresponding reference keys and storage in a PEM-formatted file.

4.5.1 Retrieve the pre-provisioned A5000 device certificates

Create a folder inside your home directory for the example certificates and keys using the following commands:

```
mkdir ~/auth_demo
```

```
cd ~/auth_demo
```

Run the following `ssscli` commands to read the device certificates and store them into a file. By default a filename with extension `.pem` and `.cer` will store the certificate in PEM format. Other extensions will store the certificate in DER format. In the following examples we use machine.pem and control_unit.pem as the certificate file names.


```
ssscli get cert F0000001 machine.pem
```

```
ssscli get cert F0000003 control_unit.pem
```

```
pi@raspberrypi:~/auth_demo $ ssscli get cert F0000001 machine.pem
Getting Certificate from KeyID = 0xF0000001
sss   :INFO :atr (Len=35)
      01 A0 00 00    03 96 04 03    E8 00 FE 02    0B 03 E8 00
      01 00 00 00    00 64 13 88    0A 00 65 53    45 30 35 31
      00 00 00
sss   :WARN :Communication channel is Plain.
sss   :WARN :!!!Not recommended for production use.!!!
Retrieved Certificate from KeyID = 0xF0000001
pi@raspberrypi:~/auth_demo $ ssscli get cert F0000003 control_unit.pem
Getting Certificate from KeyID = 0xF0000003
sss   :INFO :atr (Len=35)
      01 A0 00 00    03 96 04 03    E8 00 FE 02    0B 03 E8 00
      01 00 00 00    00 64 13 88    0A 00 65 53    45 30 35 31
      00 00 00
sss   :WARN :Communication channel is Plain.
sss   :WARN :!!!Not recommended for production use.!!!
Retrieved Certificate from KeyID = 0xF0000003
pi@raspberrypi:~/auth_demo $
```

Figure 14. Retrieve the pre-provisioned A5000 device certificats

Both certificates are stored in PEM format. These are text files containing base64 encoded data. The Linux command cat can be used to output the contents of a text file:

```
cat machine.pem
```

```
cat control_unit.pem
```

```
pi@raspberrypi:~/auth_demo $ cat machine.pem
-----BEGIN CERTIFICATE-----
MIIB0TCCAXegAwIBAgIUBABQAYycdabUYr8EMAKJrwAAAAAwCgYIKoZIzj0EAwIw
VjEXMBUGA1UECwwOUGx1ZyBhbmQgVHJ1c3QxDDAKBgNVBAoMA05YUDEtMCsGA1UE
AwwkTlhQIEludGVybWVkaWF0ZS1Db25uZWN0aXZpdHlDQXZFMjA2MB4XDTIxMTEx
NjAwMDAwMFoXDTQ2MTExMDAwMDAwMFowXzEXMBUGA1UECwwOUGx1ZyBhbmQgVHJ1
c3QxDDAKBgNVBAoMA05YUDE2MDQGA1UEAwwtRGV2Q29ubjAtMDQwMDUwMDE4QzlD
NzVBNkQ0NjJCRjA0MzAwMjg5QUYwMDAwMFkwEwYHKoZIzj0CAQYIKoZIzj0DAQcD
QgAEOMRsLXoVFWBduEHh1QhPKEFdQV22h0XTSYYjTkgiBxHMzrQbmdX0+8EVH4lm
84/I4050cnk1bXBB6KxAVcMzQqMaMBgwCQYDVR0TBAIwADALBgNVHQ8EBAMCB4Aw
CgYIKoZIzj0EAwIDSAAwRQIgT/GX4IjvbEsafelESIROttTDfz1oZjx4NxCU+3w1
BrwCIQDc+U8amM4QCPJfBya0C5/+LXU9SypEdqs1bLf/ymGs8g==
-----END CERTIFICATE-----
pi@raspberrypi:~/auth_demo $ cat control_unit.pem
-----BEGIN CERTIFICATE-----
MIIB0jCCAXegAwIBAgIUBABQAYycdabUYr8EMAKJrwAAAAEwCgYIKoZIzj0EAwIw
VjEXMBUGA1UECwwOUGx1ZyBhbmQgVHJ1c3QxDDAKBgNVBAoMA05YUDEtMCsGA1UE
AwwkTlhQIEludGVybWVkaWF0ZS1Db25uZWN0aXZpdHlDQXZFMjA2MB4XDTIxMTEx
NjAwMDAwMFoXDTQ2MTExMDAwMDAwMFowXzEXMBUGA1UECwwOUGx1ZyBhbmQgVHJ1
c3QxDDAKBgNVBAoMA05YUDE2MDQGA1UEAwwtRGV2Q29ubjEtMDQwMDUwMDE4QzlD
NzVBNkQ0NjJCRjA0MzAwMjg5QUYwMDAwMFkwEwYHKoZIzj0CAQYIKoZIzj0DAQcD
QgAEBBWwwV9gnuDFrXwmZp/WJv7k78Vv0dmVjPqUsaPAz6AfYjVak+Cmr8CeHOHq
lU/bCUZFo2yILSOHwEwBTkIgwqMaMBgwCQYDVR0TBAIwADALBgNVHQ8EBAMCB4Aw
CgYIKoZIzj0EAwIDSQAwRgIhAPrFeEOVlYgyI+0fxmm/E4tux+NmUHH05CUAgnPz
zDqzAiEA5HJ3lAY6eQzWgR/0hoXWdZHz9bhwQRGS2n3jqGE4dmU=
-----END CERTIFICATE-----
pi@raspberrypi:~/auth_demo $
```

Figure 15. Device certificats in PEM format

The `x509` Openssl command can be used to display the contents of a certificate in human readable form (`-text` switch). The `-noout` switch reduces the output by not printing the base64 encoded certificate itself.

```
openssl x509 -noout -text -in machine.pem
```

```
pi@raspberrypi:~/auth_demo $ openssl x509 -noout -text -in machine.pem
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number:
            04:00:50:01:8c:9c:75:a6:d4:62:bf:04:30:02:89:af:00:00:00:00
        Signature Algorithm: ecdsa-with-SHA256
        Issuer: OU = Plug and Trust, O = NXP, CN = NXP Intermediate-ConnectivityCAvE206
        Validity
            Not Before: Nov 16 00:00:00 2021 GMT
            Not After : Nov 10 00:00:00 2046 GMT
        Subject: OU = Plug and Trust, O = NXP, CN = DevConn0-040050018C9C75A6D462BF04300289AF0000
        Subject Public Key Info:
            Public Key Algorithm: id-ecPublicKey
                Public-Key: (256 bit)
                pub:
                    04:38:c4:6c:2d:7a:15:15:60:5d:b8:41:e1:d5:08:
                    4f:28:41:5d:41:5d:b6:87:45:d3:49:86:23:4e:48:
                    22:07:11:cc:ce:b4:1b:99:d5:f4:fb:c1:15:1f:89:
                    66:f3:8f:c8:e0:ee:4e:72:79:35:6d:70:41:e8:ac:
                    40:55:c3:33:42
                ASN1 OID: prime256v1
                NIST CURVE: P-256
        X509v3 extensions:
            X509v3 Basic Constraints:
                CA:FALSE
            X509v3 Key Usage:
                Digital Signature
    Signature Algorithm: ecdsa-with-SHA256
         30:45:02:20:4f:f1:97:e0:88:ef:6c:4b:1a:7d:e9:44:48:84:
         4e:b6:d4:c3:7f:3d:68:66:3c:78:37:10:94:fb:7c:35:06:bc:
         02:21:00:dc:f9:4f:1a:98:ce:10:08:f2:5f:07:26:b4:0b:9f:
         fe:2d:75:3d:4b:2a:44:76:ab:35:6c:b7:ff:ca:61:ac:f2
pi@raspberrypi:~/auth_demo $
```

Figure 16. Content of the machine certificate

```
openssl x509 -noout -text -in control_unit.pem
```

```
pi@raspberrypi:~/auth_demo $ openssl x509 -noout -text -in control_unit.pem
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number:
            04:00:50:01:8c:9c:75:a6:d4:62:bf:04:30:02:89:af:00:00:00:01
        Signature Algorithm: ecdsa-with-SHA256
        Issuer: OU = Plug and Trust, O = NXP, CN = NXP Intermediate-ConnectivityCAvE206
        Validity
            Not Before: Nov 16 00:00:00 2021 GMT
            Not After : Nov 10 00:00:00 2046 GMT
        Subject: OU = Plug and Trust, O = NXP, CN = DevConn1-040050018C9C75A6D462BF04300289AF0000
        Subject Public Key Info:
            Public Key Algorithm: id-ecPublicKey
                Public-Key: (256 bit)
                pub:
                    04:04:15:b0:c1:5f:60:9e:e0:c5:ad:7c:26:66:9f:
                    d6:26:fe:e4:ef:c5:6f:d1:d9:95:8c:fa:94:b1:a3:
                    c0:cf:a0:1f:62:35:5a:93:e0:a6:af:c0:9e:1c:e1:
                    ea:95:4f:db:09:46:45:a3:6c:88:2d:23:87:c0:4c:
                    01:4e:42:20:c2
                ASN1 OID: prime256v1
                NIST CURVE: P-256
        X509v3 extensions:
            X509v3 Basic Constraints:
                CA:FALSE
            X509v3 Key Usage:
                Digital Signature
    Signature Algorithm: ecdsa-with-SHA256
         30:46:02:21:00:fa:c5:78:43:95:95:88:32:23:ed:1f:c6:69:
         bf:13:8b:6e:c7:e3:66:50:71:f4:e4:25:00:82:73:f3:cc:3a:
         b3:02:21:00:e4:72:77:94:06:3a:79:0c:d6:81:1f:f4:86:85:
         d6:75:91:f3:f5:b8:70:41:11:92:da:7d:e3:a8:61:38:76:65
pi@raspberrypi:~/auth_demo $
```

Figure 17. Content of the control unit certificate

4.5.2 Retrieve the pre-provisioned A5000 device certificates public keys

The ECC public keys are required for the ECC verify operation. The ECC public keys can be extracted from the corresponding certificate using the OpenSSL command-line tool or with the help of the ssscli tool. In this chapter the ssscli tool is used.

```
ssscli get ecc pub --format PEM 0xF0000000 machine_pub_key.pem
```

```
ssscli get ecc pub --format PEM 0xF0000002
control_unit_pub_key.pem
```

```
pi@raspberrypi:~/auth_demo $ ssscli get ecc pub --format PEM 0xF0000000 machine_pub_key.pem
Getting ECC Public Key from KeyID = 0xF0000000
sss   :INFO :atr (Len=35)
      01 A0 00 00    03 96 04 03    E8 00 FE 02    0B 03 E8 00
      01 00 00 00    00 64 13 88    0A 00 65 53    45 30 35 31
      00 00 00
sss   :WARN :Communication channel is Plain.
sss   :WARN :!!!Not recommended for production use.!!!
Retrieved ECC Public Key from KeyID = 0xF0000000
pi@raspberrypi:~/auth_demo $ ssscli get ecc pub --format PEM 0xF0000002 control_unit_pub_key.pem
Getting ECC Public Key from KeyID = 0xF0000002
sss   :INFO :atr (Len=35)
      01 A0 00 00    03 96 04 03    E8 00 FE 02    0B 03 E8 00
      01 00 00 00    00 64 13 88    0A 00 65 53    45 30 35 31
      00 00 00
sss   :WARN :Communication channel is Plain.
sss   :WARN :!!!Not recommended for production use.!!!
Retrieved ECC Public Key from KeyID = 0xF0000002
pi@raspberrypi:~/auth_demo $
```

Figure 18. Retrieve the pre-provisioned A5000 device certificate's public keys

We use again the Linux command `cat` to display the both PEM formatted public keys:

```
cat machine_pub_key.pem
```

```
cat control_unit_pub_key.pem
```

```
pi@raspberrypi:~/auth_demo $ cat machine_pub_key.pem
-----BEGIN PUBLIC KEY-----
MFkwEwYHKoZIzj0CAQYIKoZIzj0DAQcDQgAEOMRsLXoVFWBduEHh1QhPKEFdQV22
h0XTSYYjTkgiBxHMzrQbmdX0+8EVH4lm84/I405Ocnk1bXBB6KxAVcMzQg==
-----END PUBLIC KEY-----
pi@raspberrypi:~/auth_demo $ cat control_unit_pub_key.pem
-----BEGIN PUBLIC KEY-----
MFkwEwYHKoZIzj0CAQYIKoZIzj0DAQcDQgAEBBWwwV9gnuDFrXwmZp/WJv7k78Vv
0dmVjPqUsaPAz6AfYjVak+Cmr8CeHOHqlU/bCUZFo2yILSOHwEwBTkIgwg==
-----END PUBLIC KEY-----
pi@raspberrypi:~/auth_demo $
```

Figure 19. Device public keys in PEM format

The `x509` OpenSSL command also supports to display the public keys contents:

```
openssl ec -pubin -in machine_pub_key.pem -text
```

```
openssl ec -pubin -in control_unit_pub_key.pem -text
```

```
pi@raspberrypi:~/auth_demo $ openssl ec -pubin -in machine_pub_key.pem -text
read EC key
Public-Key: (256 bit)
pub:
    04:38:c4:6c:2d:7a:15:15:60:5d:b8:41:e1:d5:08:
    4f:28:41:5d:41:5d:b6:87:45:d3:49:86:23:4e:48:
    22:07:11:cc:ce:b4:1b:99:d5:f4:fb:c1:15:1f:89:
    66:f3:8f:c8:e0:ee:4e:72:79:35:6d:70:41:e8:ac:
    40:55:c3:33:42
ASN1 OID: prime256v1
NIST CURVE: P-256
writing EC key
-----BEGIN PUBLIC KEY-----
MFkwEwYHKoZIzj0CAQYIKoZIzj0DAQcDQgAEOMRsLXoVFWBduEHh1QhPKEFdQV22
h0XTSYYjTkgiBxHMzrQbmdX0+8EVH4lm84/I405Ocnk1bXBB6KxAVcMzQg==
-----END PUBLIC KEY-----
pi@raspberrypi:~/auth_demo $ openssl ec -pubin -in control_unit_pub_key.pem -text
read EC key
Public-Key: (256 bit)
pub:
    04:04:15:b0:c1:5f:60:9e:e0:c5:ad:7c:26:66:9f:
    d6:26:fe:e4:ef:c5:6f:d1:d9:95:8c:fa:94:b1:a3:
    c0:cf:a0:1f:62:35:5a:93:e0:a6:af:c0:9e:1c:e1:
    ea:95:4f:db:09:46:45:a3:6c:88:2d:23:87:c0:4c:
    01:4e:42:20:c2
ASN1 OID: prime256v1
NIST CURVE: P-256
writing EC key
-----BEGIN PUBLIC KEY-----
MFkwEwYHKoZIzj0CAQYIKoZIzj0DAQcDQgAEBBWwwV9gnuDFrXwmZp/WJv7k78Vv
0dmVjPqUsaPAz6AfYjVak+Cmr8CeHOHqlU/bCUZFo2yILSOHwEwBTkIgwg==
-----END PUBLIC KEY-----
pi@raspberrypi:~/auth_demo $
```

Figure 20. Content of the device public keys

4.5.3 Create the reference key files for the OpenSSL engine

As already described above, the ECC private keys are securely stored inside the A5000 and cannot be read out like the public certificate or public key. To be able to delegate a private crypto operation like an ECC signature generation to the A5000 we need to generate a reference key. Later we use the reference key instead of the private key for OpenSSL operations.

The following two commands are generating a "machine" and "control unit" reference key.

```
ssscli refpem ecc pair 0xF0000000 machine_ref_key.pem
```

```
ssscli refpem ecc pair 0xF0000002 control_unit_ref_key.pem
```



```
pi@raspberrypi:~/auth_demo $ ssscli refpem ecc pair 0xF0000000 machine_ref_key.pem
sss   :INFO :atr (Len=35)
      01 A0 00 00    03 96 04 03    E8 00 FE 02    0B 03 E8 00
      01 00 00 00    00 64 13 88    0A 00 65 53    45 30 35 31
      00 00 00
sss   :WARN :Communication channel is Plain.
sss   :WARN :!!!Not recommended for production use.!!!
Created reference key for ECC Pair from KeyID = 0xF0000000
pi@raspberrypi:~/auth_demo $ ssscli refpem ecc pair 0xF0000002 control_unit_ref_key.pem
sss   :INFO :atr (Len=35)
      01 A0 00 00    03 96 04 03    E8 00 FE 02    0B 03 E8 00
      01 00 00 00    00 64 13 88    0A 00 65 53    45 30 35 31
      00 00 00
sss   :WARN :Communication channel is Plain.
sss   :WARN :!!!Not recommended for production use.!!!
Created reference key for ECC Pair from KeyID = 0xF0000002
pi@raspberrypi:~/auth_demo $
```

Figure 21. Create the reference key files for the OpenSSL engine

The `ssscli` commands above are storing the reference keys in PEM format.

```
cat machine_ref_key.pem
```

```
cat control_unit_ref_key.pem
```

```
pi@raspberrypi:~/auth_demo $ cat machine_ref_key.pem
-----BEGIN EC PRIVATE KEY-----
MHcCAQEEIBAAAAAAAAAAAAAAAAAAAAAAPAAAAClprW2paa1thAAoAoGCCqGSM49
AwEHoUQDQgAEOMRsLXoVFWBduEHh1QhPKEFdQV22h0XTSYYjTkgiBxHMzrQbmdX0
+8EVH4lm84/I405Ocnk1bXBB6KxAVcMzQg==
-----END EC PRIVATE KEY-----
pi@raspberrypi:~/auth_demo $ cat control_unit_ref_key.pem
-----BEGIN EC PRIVATE KEY-----
MHcCAQEEIBAAAAAAAAAAAAAAAAAAAAAAPAAAAKlprW2paa1thAAoAoGCCqGSM49
AwEHoUQDQgAEBBWwwV9gnuDFrXwmZp/WJv7k78Vv0dmVjPqUsaPAz6AfYjVak+Cm
r8CeHOHqlU/bCUZFo2yILSOHwEwBTkIgwg==
-----END EC PRIVATE KEY-----
pi@raspberrypi:~/auth_demo $
```

Figure 22. Reference private keys in PEM format

In the first glance, the reference key looks like as any other private key, therefore it is required to use OpenSSL to display the details:

```
openssl ec -in machine_ref_key.pem -text
```

```
openssl ec -in control_unit_ref_key.pem -text
```

```
pi@raspberrypi:~/auth_demo $ openssl ec -in machine_ref_key.pem -text
read EC key
Private-Key: (256 bit)
priv:
    10:00:00:00:00:00:00:00:00:00:00:00:00:00:00:
    00:00:00:f0:00:00:00:a5:a6:b5:b6:a5:a6:b5:b6:
    10:00
             machine private ECC key - object ID = 0xF0000000
pub:
    04:38:c4:6c:2d:7a:15:15:60:5d:b8:41:e1:d5:08:
    4f:28:41:5d:41:5d:b6:87:45:d3:49:86:23:4e:48:
    22:07:11:cc:ce:b4:1b:99:d5:f4:fb:c1:15:1f:89:
    66:f3:8f:c8:e0:ee:4e:72:79:35:6d:70:41:e8:ac:
    40:55:c3:33:42
ASN1 OID: prime256v1
NIST CURVE: P-256
writing EC key
-----BEGIN EC PRIVATE KEY-----
MHcCAQEEIBAAAAAAAAAAAAAAAAAAAAAAAPAAAAClprW2paa1thAAoAoGCCqGSM49
AwEHoUQDQgAEOMRsLXoVFWBduEHh1QhPKEFdQV22h0XTSYYjTkgiBxHMzrQbmdX0
+8EVH4lm84/I405Ocnk1bXBB6KxAVcMzQg==
-----END EC PRIVATE KEY-----
pi@raspberrypi:~/auth_demo $ openssl ec -in control_unit_ref_key.pem -text
read EC key
Private-Key: (256 bit)
priv:
    10:00:00:00:00:00:00:00:00:00:00:00:00:00:00:
    00:00:00:f0:00:00:02:a5:a6:b5:b6:a5:a6:b5:b6:
    10:00
             control unit private ECC key - object ID = 0xF0000002
pub:
    04:04:15:b0:c1:5f:60:9e:e0:c5:ad:7c:26:66:9f:
    d6:26:fe:e4:ef:c5:6f:d1:d9:95:8c:fa:94:b1:a3:
    c0:cf:a0:1f:62:35:5a:93:e0:a6:af:c0:9e:1c:e1:
    ea:95:4f:db:09:46:45:a3:6c:88:2d:23:87:c0:4c:
    01:4e:42:20:c2
ASN1 OID: prime256v1
NIST CURVE: P-256
writing EC key
-----BEGIN EC PRIVATE KEY-----
MHcCAQEEIBAAAAAAAAAAAAAAAAAAAAAAAPAAAAKlprW2paa1thAAoAoGCCqGSM49
AwEHoUQDQgAEBBWwwV9gnuDFrXwmZp/WJv7k78Vv0dmVjPqUsaPAz6AfYjVak+Cm
r8CeHOHqlU/bCUZFo2yILSOHwEwBTkIgwg==
-----END EC PRIVATE KEY-----
pi@raspberrypi:~/auth_demo $
```

Figure 23. Content of the reference private keys

Instead of a real private ECC device key the reference key contains mainly the A5000 private key object ID. The remaining bytes are containing a 64-bit "magic number" (always `0xA5A6B5B6A5A6B5B6`). The Plug & Trust Middleware documentation provides a detailed description of the reference key format.

The NXP OpenSSL engine uses this "magic number" to distinguish a reference key from a real private key. In case a reference key is passed to the OpenSSL API or command-line tool the NXP OpenSSL engine will invoke the A5000 to perform the private crypto operation.

4.6 Chain of trust of the pre-provisioned device certificates

A certificate is a digital document that contains a public key and additional information about the entity associated with it. A certificate also includes a digital signature from the certificate issuer. In case of the pre-provisioned A5000 certificates the certification issuer is NXP. The image below shows the complete certification chain of the pre-provisioned device certificates.

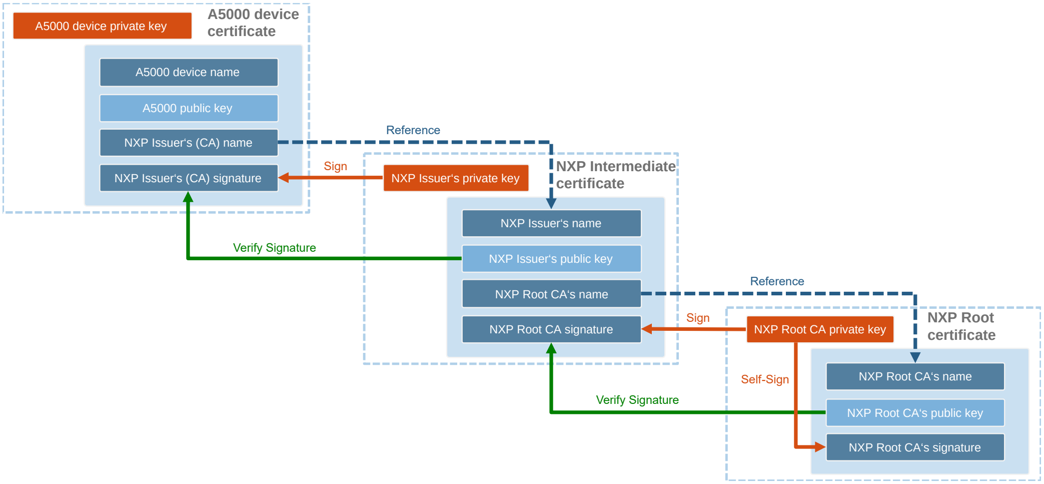


Figure 24. Certification chain of the pre-provisioned A5000 device certificates

All pre-provisioned A5000 device certificates are signed with the associated private key of the NXP intermediate certificate. To verify the validity of the pre-provisined device certificates we need to download the intermediate certificate.

The NXP intermediate certificate can be downloaded via the following link: https://www.gp-ca.nxp.com/CA/getCA?caid=63709315060022.

The Linux command wegt can be used to download the NXP intermediate certificate. The -O parameter is used to specify the filename.

```
wget https://www.gp-ca.nxp.com/CA/getCA?caid=63709315060022 -O
nxp_a5000_intermediate_ca.crt
```

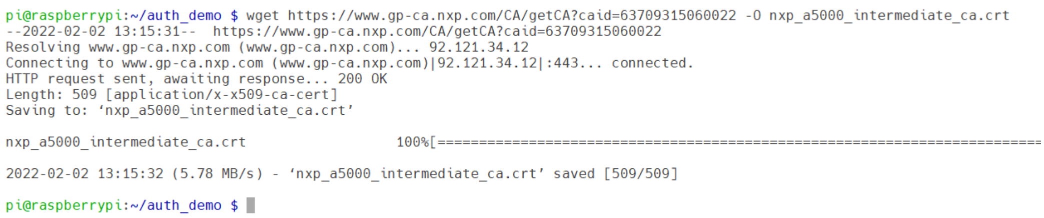


Figure 25. Download the NXP intermediate certificate

The file `nxp_a5000_intermediate_ca.crt` contains NXP intermediate certificate in DER format. For the following OpenSSL command-line examples it is required to convert the certificate into the PEM formatted certification file (`nxp_a5000_intermediate_ca.pem`). This step can be performed using the following OpenSSL command:

```
openssl x509 -in nxp_a5000_intermediate_ca.crt -inform der -out
nxp_a5000_intermediate_ca.pem -outform pem
```

```
cat nxp_a5000_intermediate_ca.pem
```



```
pi@raspberrypi:~/auth_demo $ openssl x509 -in nxp_a5000_intermediate_ca.crt -inform der -out nxp_a5000_intermediate_ca.pem -outform pem
pi@raspberrypi:~/auth_demo $ cat nxp_a5000_intermediate_ca.pem
```

```
-----BEGIN CERTIFICATE-----
MIIB+TCCAVigAwIBAgIBBTAMBggqhkjOPQQDAgUAMEExFzAVBgNVBAsMDlBsdWcg
YW5kIFRydXN0MQwwCgYDVQQKDANOWFAxGDAWBgNVBAMMD05YUCBSb290Q0F2RTUw
NjAiGA8yMDIxMTIxNTExNTYxNloYDzIwNzExMjE1MTE1NjE2WjBWMRcwFQYDVQQL
DA5QbHVnIGFuZCBUcnVzdDEMMAoGA1UECgwDTlhQMS0wKwYDVQQDDCROWFAgSW50
ZXJtZWRpYXRlLUNvbm5lY3Rpdml0eUNBdkUyMDYwWTATBgcqhkjOPQIBBggqhkjO
PQMBBwNCAAQT7b6qNuRGM2is5psP4lur2xQTjMJToFhzRv26BzXAmgviex3pFdzs
FUCA2NBY56M9D38+ZIZTorF0hhqOKasgoyYwJDASBgNVHRMBAf8ECDAGAQH/AgEA
MA4GA1UdDwEB/wQEAwIBBjAMBggqhkjOPQQDAgUAA4GMADCBiAJCAWxjV7Sw4nZY
3vvT+sMuzc2eHciHwRDUHw112DfM1x+V/gr/3fCLISipjV5YtkKGpnRMFwU+gjTD
pzC1adR16OkPAkIA0rVA23/tonWoi0k3fGcjimIOGr9pQHEYhM5yZb97zNNxwyrK
WDdGA2lNc2QxZgPPkUkzBQS8FgJflkmHLl4LQk0=
-----END CERTIFICATE-----
```

```
pi@raspberrypi:~/auth_demo $
```

Figure 26. Convert the NXP intermediate certificate file nxp_a5000_intermediate_ca.crt" into a PEM formatted file

The NXP intermediate certificate is signed by a NXP root certificate. To be able to verify the validity of the NXP intermediate certificate you need also to download the NXP root certificate.

The NXP root certificate can be downloaded via the following link: https://www.gp-ca.nxp.com/CA/getCA?caid=63709315050010.

We can use again the Linux command `wegt` to download the certificate:

```
wget https://www.gp-ca.nxp.com/CA/getCA?caid=63709315050010 -O
nxp_a5000_root_ca.crt
```

```
pi@raspberrypi:~/auth_demo $ wget https://www.gp-ca.nxp.com/CA/getCA?caid=63709315050010 -O nxp_a5000_root_ca.crt
--2022-02-02 13:21:22--  https://www.gp-ca.nxp.com/CA/getCA?caid=63709315050010
Resolving www.gp-ca.nxp.com (www.gp-ca.nxp.com)... 92.121.34.12
Connecting to www.gp-ca.nxp.com (www.gp-ca.nxp.com)|92.121.34.12|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 552 [application/x-x509-ca-cert]
Saving to: 'nxp_a5000_root_ca.crt'

nxp_a5000_root_ca.crt                 100%[================================================================

2022-02-02 13:21:23 (5.90 MB/s) - 'nxp_a5000_root_ca.crt' saved [552/552]

pi@raspberrypi:~/auth_demo $
```

Figure 27. Download the NXP root certificate

Finally we also convert the NXP root certificate into a PEM formatted certification file (`nxp_a5000_root_ca.pem`).

```
openssl x509 -in nxp_a5000_root_ca.crt -inform der -out
nxp_a5000_root_ca.pem -outform pem
```

```
cat nxp_a5000_root_ca.pem
```

```
pi@raspberrypi:~/auth_demo $ openssl x509 -in nxp_a5000_root_ca.crt -inform der -out nxp_a5000_root_ca.pem -outform pem
pi@raspberrypi:~/auth_demo $ cat nxp_a5000_root_ca.pem
```

```
-----BEGIN CERTIFICATE-----
MIICJDCCAYOgAwIBAgIBBTAMBggqhkjOPQQDBAUAMEExFzAVBgNVBAsMDlBsdWcg
YW5kIFRydXN0MQwwCgYDVQQKDANOWFAxGDAWBgNVBAMMD05YUCBSb290Q0F2RTUw
NjAiGA8yMDIxMTIxNTExNDMyNloYDzIwODExMjE1MTE0MzI2WjBBMRcwFQYDVQQL
DA5QbHVnIGFuZCBUcnVzdDEMMAoGA1UECgwDTlhQMRgwFgYDVQQDDA9OWFAgUm9v
dENBdkU1MDYwgZswEAYHKoZIzj0CAQYFK4EEACMDgYYABABiD63wcZccqOcxdyj+
uW3PLVyE1/oJgl/wMvdgEhmywny3HOoZGsWfA967gZqydcj6n/AYsc20rVOEGsBN
YUP/HwFmoVPpLTBwfWlu0JxvnEXuW7lj9AEenRG+Bo5AjnZX/QlY6BkKhzjPcc62
Jze9Bt0h6gr67RZHBPkL3leri0M466MjMCEwDwYDVR0TAQH/BAUwAwEB/zAOBgNV
HQ8BAf8EBAMCAQYwDAYIKoZIzj0EAwQFAAOBjAAwgYgCQgEIZDkAOF43uv4UI7XE
fnZQuMFnYsN6XaZmvo3R/oW0Jqa4+xTGjIIerXubsRiC07UvaKl8uwj4WTe90K+L
trZr5wJCAdAqKemLF6NjLC5p4wExu4ciPOw9K5cFcH83rYU9NSQTkchvxbe/wjX8
HMLekqRYwMapF/dF0vtN/TCNmgE+wPBo
-----END CERTIFICATE-----
```

```
pi@raspberrypi:~/auth_demo $
```

Figure 28. Convert the NXP root certificate file "nxp_a5000_root_ca.crt"into a PEM formatted file

4.7 Mutual authentication flow

As already described in Section 3 the authentication flow consists of a mutual authentication procedure. First, the machine will authenticate the control unit. If the machine was successfully authenticated, the control unit will authenticate the machine.

4.7.1 Control unit authentication

The authentication of the control unit consists of two steps:

- Step 1: Control unit device certificate validation
- Step 2: Proof of control unit private key possession

The example below will demonstrate the basic principle of the control unit authentication flow as show in the figure below using the OpenSSL command-line tools .

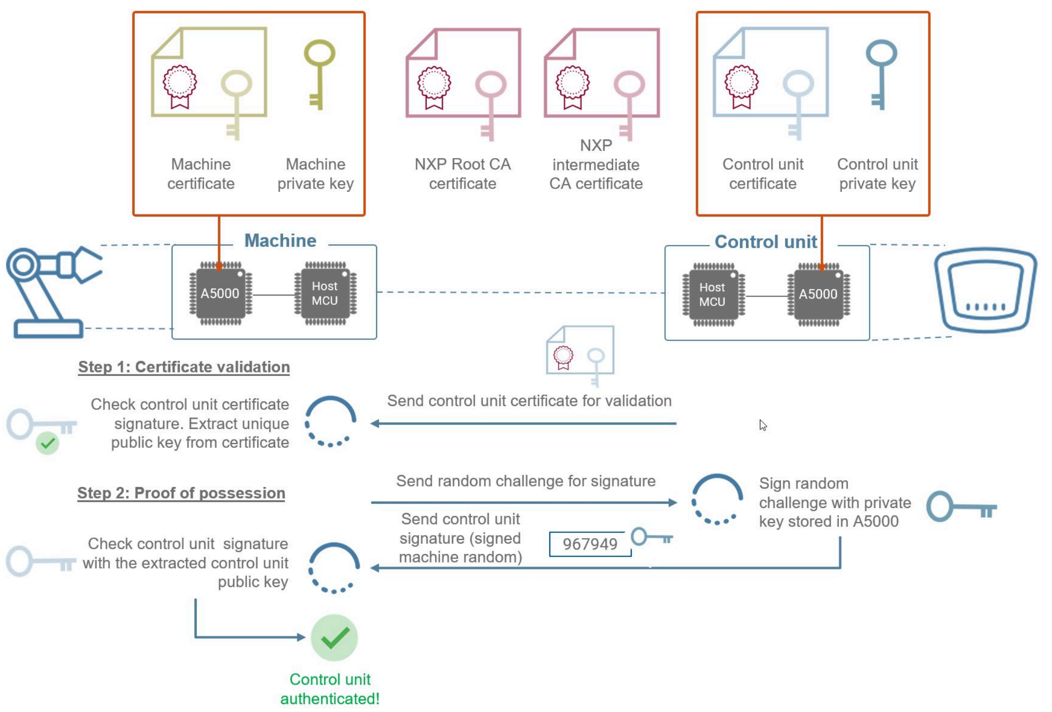


Figure 29. Control unit authentication flow

4.7.1.1 Step 1: Control unit device certificate validation

The first step the control unit sends the control unit certificate (`control_unit.pem`) to the machine for validating the certificate. The OpenSSL `verify` command-line tools allows the validation of a certification chain. It is required to provide OpenSSL the NXP A5000 root CA and the NXP A5000 intermediate CA and the A5000 device certificate to be validated:

```
openssl verify -CAfile nxp_a5000_root_ca.pem -untrusted
nxp_a5000_intermediate_ca.pem control_unit.pem
```

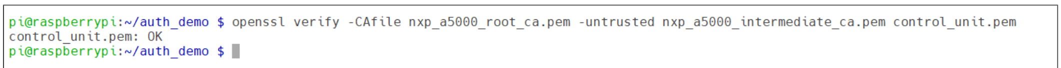


Figure 30. OpenSSL - Verify control unit device certificate

Note: *We assume the NXP root and intermediate CA are already stored in the machine and control unit.*

Note: *To simplify the example we do not use the A5000 for validating the control unit certificate, because the keys of the NXP root and intermediate CA are not stored inside the A5000 device.*

The control unit certificate is valid in case OpenSSL returns `OK`. This also means, that the public key included in the control unit certificate can be trusted.

4.7.1.2 Step 2: Proof of control unit private key possession

In this step, the control unit must prove that it is in possession of the ECC private key.

- For this purpose, the machine uses the A5000 to generate a 16-byte random number and sends this number to control unit.
- Next the control unit uses A5000 to sign the received random number, using the private ECC key securely stored inside the A5000.
- The ECC signature (signed random number) is returned to the machine.
- The machine verifies the signature with the control unit public key. The control unit is authenticated in case of successful signature verification.

The following OpenSSL command generates 8-byte random number in HEX format. Because we did not specify to use the A5000 OpenSSL engine so random numbers are generated by OpenSSL in software.

```
openssl rand -hex 8
```

Figure 31. OpenSSL - Random numbers generated by OpenSSL in software

The Plug & Trust Middleware supports the A5000 Secure Authenticator and the SE05x Secure Element. Both product families are using the same API and the same OpenSSL engine. The NXP Plug & Trust middleware OpenSSL engine is located in the following directory:

```
/usr/local/lib/libsss_engine.so
```

The corresponding OpenSSL configuration file is located in:

```
~/se_mw/simw-top/demos/linux/common/openssl_sss_se050.cnf
```

With the help of following the Linux command we can display the relevant default setting:

```
tail ~/se_mw/simw-top/demos/linux/common/openssl11_sss_se050.cnf
-n 12
```



```
pi@raspberrypi:~/auth_demo $ tail ~/se_mw/simw-top/demos/linux/common/openssl11_sss_se050.cnf -n 12
[nxp_engine]
engines = engine_section

[engine_section]
e4sss_se050 = e4sss_se050_section

[e4sss_se050_section]
engine_id = e4sss
dynamic_path = /usr/local/lib/libsss_engine.so
init = 1
default_algorithms = RAND,RSA,EC

pi@raspberrypi:~/auth_demo $
```

Figure 32. Plug & Trust Middleware OpenSSL engine default configuration

Note: *The A5000 does not support RSA, there it is recommended to remove the entry RSA from the default algorithmus entry.*

We can keep the default settings unmodified. To overrule the default OpenSSL configuration, we can temporally assign the path to the **`openssl11_sss_se050.cnf`** file by setting the Linux environment variable `OPENSSL_CONF`. This step is performed with the help of the shell's `export` command.

```
export OPENSSL_CONF=~/se_mw/simw-top/demos/linux/common/
openssl11_sss_se050.cnf
```

Now we can use the same OpenSSL command to delegate the random numbers generation to the A5000. The console output includes P&T MW default log messages.

```
openssl rand -hex 8
```

```
pi@raspberrypi:~/auth_demo $ export OPENSSL_CONF=~/se_mw/simw-top/demos/linux/common/openssl11_sss_se050.cnf
pi@raspberrypi:~/auth_demo $ openssl rand -hex 8
ssse-flw: EmbSe_Init(): Entry
App    :INFO :If you want to over-ride the selection, use ENV=EX_SSS_BOOT_SSS_PORT or pass in command line arguments.
sss    :INFO :atr (Len=35)
       01 A0 00 00    03 96 04 03    E8 00 FE 02    0B 03 E8 00
       01 00 00 00    00 64 13 88    0A 00 65 53    45 30 35 31
       00 00 00
sss    :WARN :Communication channel is Plain.
sss    :WARN :!!!Not recommended for production use.!!!
ssse-flw: Version: 1.0.5
ssse-flw: EmbSe_Init(): Exit
ssse-flw: EmbSe_Rand invoked requesting 8 random bytes
2bedc125612c09eb random numbers generated by A5000
ssse-flw: EmbSe_Finish(): Entry
ssse-flw: EmbSe_Finish(): Exit
ssse-flw: EmbSe_Destroy(): Entry
pi@raspberrypi:~/auth_demo $
```

Figure 33. OpenSSL - Random number generated by A5000

Next in our example, the machine generates a 256-byte random number and stores it into a text file. The random number is send to the control unit.

```
openssl rand -out machine_random.txt -hex 256
```

```
cat machine_random.txt
```

```
pi@raspberrypi:~/auth_demo $ openssl rand -out machine_random.txt -hex 256
ssse-flw: EmbSe_Init(): Entry
App   :INFO :If you want to over-ride the selection, use ENV=EX_SSS_BOOT_SSS_PORT or pass
sss   :INFO :atr (Len=35)
      01 A0 00 00    03 96 04 03    E8 00 FE 02    0B 03 E8 00
      01 00 00 00    00 64 13 88    0A 00 65 53    45 30 35 31
      00 00 00
sss   :WARN :Communication channel is Plain.
sss   :WARN :!!!Not recommended for production use.!!!
ssse-flw: Version: 1.0.5
ssse-flw: EmbSe_Init(): Exit
ssse-flw: EmbSe_Rand invoked requesting 256 random bytes
ssse-flw: EmbSe_Finish(): Entry
ssse-flw: EmbSe_Finish(): Exit
ssse-flw: EmbSe_Destroy(): Entry
pi@raspberrypi:~/auth_demo $ cat machine_random.txt
ef6021c9a6d1c54284bd3823ee8f7e6d885c36873094b3104960577e3b5464b94f0fe117b27d376d24e8f4ba77
bdc443a61535bf80e1717f3d780fa712796128aba8fa91acc7124a3d88fecd33afb9a27f7ce8eac51882b21d36
1d8adfc539408ca04b97ee467421efee30b31df05f36eb8704934f97447bdae408d3173a34f4a3b1751d3a7a83
pi@raspberrypi:~/auth_demo $
```

Figure 34. OpenSSL - A5000 random numbers are stored in a text file

The control unit uses the A5000 to generate the ECC signature using standard OpenSSL commands. This is performed by providing a control unit reference key (`control_unit_ref_key.pem`) instead of a private key. The signature is stored in the `sig_machine_random.sha256` in binary format.

```
openssl dgst -sha256 -sign control_unit_ref_key.pem -out
control_unit_signature.sha256 machine_random.txt
```

```
pi@raspberrypi:~/auth_demo $ openssl dgst -sha256 -sign control_unit_ref_key.pem -out control_unit_signature.sha256 machine_random.txt
ssse-flw: EmbSe_Init(): Entry
App   :INFO :If you want to over-ride the selection, use ENV=EX_SSS_BOOT_SSS_PORT or pass in command line arguments.
sss   :INFO :atr (Len=35)
      01 A0 00 00    03 96 04 03    E8 00 FE 02    0B 03 E8 00
      01 00 00 00    00 64 13 88    0A 00 65 53    45 30 35 31
      00 00 00
sss   :WARN :Communication channel is Plain.
sss   :WARN :!!!Not recommended for production use.!!!
ssse-flw: Version: 1.0.5
ssse-flw: EmbSe_Init(): Exit
ssse-dbg: Using keyId=0xF0000002
ssse-dbg: shaAlgo: 771
ssse-flw: SSS based sign (keyId=0xF0000002, dgstLen=32)
ssse-flw: SSS based sign called successfully (sigDERLen=71)
ssse-flw: EmbSe_ECDSA_Do_Sign success.
ssse-flw: EmbSe_Finish(): Entry
ssse-flw: EmbSe_Finish(): Exit
ssse-flw: EmbSe_Destroy(): Entry
pi@raspberrypi:~/auth_demo $
```

Figure 35. OpenSSL - The A5000 signs the random numbers with the private ECC key stored inside the A5000

We can use the following Linux command to display the binary signature value.

```
xxd -c 16 -g 1 -u control_unit_signature.sha256
```

```
pi@raspberrypi:~/auth_demo $ xxd -c 16 -g 1 -u control_unit_signature.sha256
00000000: 30 45 02 20 76 D0 FE B1 39 04 A4 62 E4 A4 93 2B  0E. v...9..b...+
00000010: E4 54 43 2F 55 8A 22 D1 44 B3 36 8E FB 10 DB 61  .TC/U.".D.6....a
00000020: F7 F4 60 B1 02 21 00 EC AF EE CE D6 3B 29 72 62  ..`..!......;)rb
00000030: 0C 8B D0 EB DC 25 0E 8B 82 44 EE D2 BA F2 4C B3  .....%...D....L.
00000040: A8 2A F4 34 48 7F 10                             .*.4H..
pi@raspberrypi:~/auth_demo $
```

Figure 36. Control unit signature

The machine extracts the unique control unit public key from certificate using the following OpenSSL command:

```
openssl x509 -in control_unit.pem -pubkey -noout >
control_unit_pub.pem
```

Finally, the machine verifies the signature with the control unit public key `control_unit_pub.pem`. Because we are using the public key of another entity, this step is performed by the OpenSSL engine in software.

```
openssl dgst -sha256 -verify control_unit_pub.pem -signature
control_unit_signature.sha256 machine_random.txt
```

```
pi@raspberrypi:~/auth_demo $ openssl x509 -in control_unit.pem -pubkey -noout > control_unit_pub.pem
pi@raspberrypi:~/auth_demo $ openssl dgst -sha256 -verify control_unit_pub.pem -signature control_unit_signature.sha256 machine_random.txt
ssse-flw: EmbSe_Init(): Entry
App   :INFO :If you want to over-ride the selection, use ENV=EX_SSS_BOOT_SSS_PORT or pass in command line arguments.
sss   :INFO :atr (Len=35)
      01 A0 00 00    03 96 04 03    E8 00 FE 02    0B 03 E8 00
      01 00 00 00    00 64 13 88    0A 00 65 53    45 30 35 31
      00 00 00
sss   :WARN :Communication channel is Plain.
sss   :WARN :!!!Not recommended for production use.!!!
ssse-flw: Version: 1.0.5
ssse-flw: EmbSe_Init(): Exit
ssse-flw: Invoking EmbSe_ECDSA_Do_Verify(..)
ssse-dbg: ====>SIGNATURE (len=71)
ssse-dbg:
30 45 02 20 76 D0 FE B1 39 04 A4 62 E4 A4 93 2B
E4 54 43 2F 55 8A 22 D1 44 B3 36 8E FB 10 DB 61
F7 F4 60 B1 02 21 00 EC AF EE CE D6 3B 29 72 62
0C 8B D0 EB DC 25 0E 8B 82 44 EE D2 BA F2 4C B3
A8 2A F4 34 48 7F 10
ssse-dbg: ====>DIGEST
4C E0 C3 A6 68 DD 3D 8E CC 63 43 73 0F C8 1D 35
4A 1C 3B 91 C3 D3 CE 4A 0B 56 C0 55 F8 A3 40 FF
ssse-flw: No matching key in Secure Element. Invoking OpenSSL API: ECDSA_do_verify.
ssse-flw: Verification by OpenSSL PASS
Verified OK
ssse-flw: EmbSe_Finish(): Entry
ssse-flw: EmbSe_Finish(): Exit
ssse-flw: EmbSe_Destroy(): Entry
pi@raspberrypi:~/auth_demo $
```

Figure 37. OpenSSL - Verify control unit signature

The control unit is authenticated in case OpenSSL returns `Verified OK`.

4.7.2 Machine authentication

The authentication of the machine also consists of two steps. In principle the steps are vice versa compared to the control unit authentication. The steps are briefly demonstrated for completeness:

- Step 1: Machine certificate validation
- Step 2: Proof of machine private key possession

The example below will demonstrate the basic principle of the machine authentication flow as shown in the figure below using the OpenSSL command-line tools .

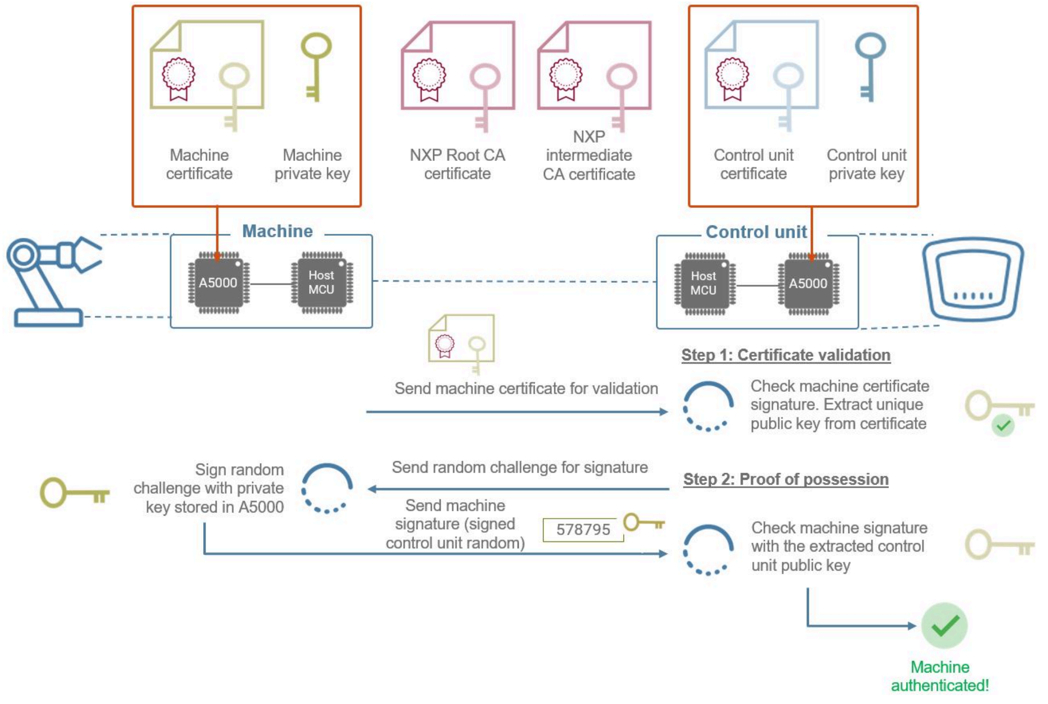


Figure 38. Control unit authentication flow

4.7.2.1 Step 1: Machine device certificate validation

The first step the machine sends the machine certificate (`machine.pem`) to the control unit for validating the certificate. We use again the OpenSSL `verify` command-line tools to validate the certification chain.

```
openssl verify -CAfile nxp_a5000_root_ca.pem -untrusted
nxp_a5000_intermediate_ca.pem machine.pem
```

```
pi@raspberrypi:~/auth_demo $ openssl verify -CAfile nxp_a5000_root_ca.pem -untrusted nxp_a5000_intermediate_ca.pem machine.pem
ssse-flw: EmbSe_Init(): Entry
App   :INFO :If you want to over-ride the selection, use ENV=EX_SSS_BOOT_SSS_PORT or pass in command line arguments.
sss   :INFO :atr (Len=35)
      01 A0 00 00    03 96 04 03    E8 00 FE 02    0B 03 E8 00
      01 00 00 00    00 64 13 88    0A 00 65 53    45 30 35 31
      00 00 00
sss   :WARN :Communication channel is Plain.
sss   :WARN :!!!Not recommended for production use.!!!
ssse-flw: Version: 1.0.5
ssse-flw: EmbSe_Init(): Exit
ssse-flw: Invoking EmbSe_ECDSA_Do_Verify(..)
ssse-dbg: ====>SIGNATURE (len=139)
ssse-dbg:
30 81 88 02 42 01 6C 63 57 B4 B0 E2 76 58 DE FB
D3 FA C3 2E CD CD 9E 1D C8 87 C1 10 D4 1F 0D 75
D8 37 CC D7 1F 95 FE 0A FF DD F0 8B 21 28 A9 8D
5E 58 B6 42 86 A6 74 4C 17 05 3E 82 34 C3 A7 30
B5 69 D4 75 E8 E9 0F 02 42 00 D2 B5 40 DB 7F ED
A2 75 A8 8B 49 37 7C 67 23 8A 62 0E 1A BF 69 40
71 18 84 CE 72 65 BF 7B CC D3 71 C3 2A CA 58 37
46 03 69 4D 73 64 31 66 03 CF 91 49 33 05 04 BC
16 02 5F 96 49 87 2E 5E 0B 42 4D
ssse-dbg: ====>DIGEST
76 6E 71 6F 6F 36 33 DD 65 7D 22 AB 1F D0 69 A6
0D ED 28 CC C1 E1 65 A6 40 3F 04 03 1F 8D 79 02
ssse-flw: No matching key in Secure Element. Invoking OpenSSL API: ECDSA_do_verify.
ssse-flw: Verification by OpenSSL PASS
ssse-flw: Invoking EmbSe_ECDSA_Do_Verify(..)
ssse-dbg: ====>SIGNATURE (len=71)
ssse-dbg:
30 45 02 20 4F F1 97 E0 88 EF 6C 4B 1A 7D E9 44
48 84 4E B6 D4 C3 7F 3D 68 66 3C 78 37 10 94 FB
7C 35 06 BC 02 21 00 DC F9 4F 1A 98 CE 10 08 F2
5F 07 26 B4 0B 9F FE 2D 75 3D 4B 2A 44 76 AB 35
6C B7 FF CA 61 AC F2
ssse-dbg: ====>DIGEST
98 31 55 B6 C6 A0 E5 63 38 51 26 5F AE 00 D2 16
B5 A9 B9 3D 58 17 5E D0 A9 B9 91 B4 A2 48 73 E9
ssse-flw: No matching key in Secure Element. Invoking OpenSSL API: ECDSA_do_verify.
ssse-flw: Verification by OpenSSL PASS
machine.pem: OK
ssse-flw: EmbSe_Finish(): Entry
ssse-flw: EmbSe_Finish(): Exit
ssse-flw: EmbSe_Destroy(): Entry
pi@raspberrypi:~/auth_demo $
```

Figure 39. OpenSSL - Verify machine certificate

Note: *We assume the NXP root and intermediate CA are already stored in the machine and control unit.*

Note: *To simplify the example we do not use the A5000 for validating the machine certificate, because the keys of the NXP root and intermediate CA are not stored inside the A5000 device.*

The machine certificate is valid in case OpenSSL returns `OK`. This also means, that the public key included in the machine certificate can be trusted.

4.7.2.2 Step 2: Proof of control unit private key possession

In this step, the machine must prove that it is in possession of the ECC private key.

Note: *We assume the Linux environment variable `OPENSSL_CONF` was already set as described in* *Section 4.7.1.2**.*

The control unit generates a 256-bytes random number and stores it into a text file. The random number is sent to the machine.

```
openssl rand -out control_unit_random.txt -hex 256
```

```
cat control_unit_random.txt
```

```
pi@raspberrypi:~/auth_demo $ openssl rand -out control_unit_random.txt -hex 256
ssse-flw: EmbSe_Init(): Entry
App   :INFO :If you want to over-ride the selection, use ENV=EX_SSS_BOOT_SSS_PORT or pass
sss   :INFO :atr (Len=35)
      01 A0 00 00    03 96 04 03    E8 00 FE 02    0B 03 E8 00
      01 00 00 00    00 64 13 88    0A 00 65 53    45 30 35 31
      00 00 00
sss   :WARN :Communication channel is Plain.
sss   :WARN :!!!Not recommended for production use.!!!
ssse-flw: Version: 1.0.5
ssse-flw: EmbSe_Init(): Exit
ssse-flw: EmbSe_Rand invoked requesting 256 random bytes
ssse-flw: EmbSe_Finish(): Entry
ssse-flw: EmbSe_Finish(): Exit
ssse-flw: EmbSe_Destroy(): Entry
pi@raspberrypi:~/auth_demo $ cat control_unit_random.txt
83ce8d10ee8b20ac3b81562a60597d9066021b906367810b9be319b5e5bb988675f4fce5c74afa1ebbd1b5b210
cce0da92af3b3b9bb9b22a732635ba2cd88854ff41a5a51afce84b56636861a09eef698a83f7c2da4741855329
576a1b8fc984f5ec0ed987784ca13aadb439e99cff547be440cccc8b3d9cef19c2a0cb6fd8a49f527b3b154d94
pi@raspberrypi:~/auth_demo $
```

Figure 40. OpenSSL - A5000 random numbers are stored in a text file

The machine uses the A5000 to generate the ECC signature. This is performed by providing a machine reference key (`machine_ref_key.pem`) instead of a private key. The signature is stored in the `sig_control_unit_random.sha256` in binary format.

```
openssl dgst -sha256 -sign machine_ref_key.pem -out
mashine_signature.sha256 control_unit_random.txt
```

```
pi@raspberrypi:~/auth_demo $ openssl dgst -sha256 -sign machine_ref_key.pem -out mashine_signature.sha256 control_unit_random.txt
ssse-flw: EmbSe_Init(): Entry
App   :INFO :If you want to over-ride the selection, use ENV=EX_SSS_BOOT_SSS_PORT or pass in command line arguments.
sss   :INFO :atr (Len=35)
      01 A0 00 00    03 96 04 03    E8 00 FE 02    0B 03 E8 00
      01 00 00 00    00 64 13 88    0A 00 65 53    45 30 35 31
      00 00 00
sss   :WARN :Communication channel is Plain.
sss   :WARN :!!!Not recommended for production use.!!!
ssse-flw: Version: 1.0.5
ssse-flw: EmbSe_Init(): Exit
ssse-dbg: Using keyId=0xF0000000
ssse-dbg: shaAlgo: 771
ssse-flw: SSS based sign (keyId=0xF0000000, dgstLen=32)
ssse-flw: SSS based sign called successfully (sigDERLen=70)
ssse-flw: EmbSe ECDSA Do Sign success.
ssse-flw: EmbSe_Finish(): Entry
ssse-flw: EmbSe_Finish(): Exit
ssse-flw: EmbSe_Destroy(): Entry
pi@raspberrypi:~/auth_demo $
```

Figure 41. OpenSSL - The A5000 signs the random numbers with the private ECC key stored inside the A5000

We can use the following Linux command to display the binary signature value.

```
xxd -c 16 -g 1 -u mashine_signature.sha256
```

```
pi@raspberrypi:~/auth_demo $ xxd -c 16 -g 1 -u mashine_signature.sha256
00000000: 30 44 02 20 1A D9 1F 44 D6 87 27 22 5B 7D E9 31  0D. ...D..'"[}.1
00000010: A3 46 3A 5D B2 95 E2 22 B9 0F 07 AC 10 6D 34 0D  .F:]..."......m4.
00000020: 09 16 FA 4E 02 20 67 E8 FF AB ED 74 62 8A CF DE  ...N. g....tb...
00000030: BB 2E AF EF BE 93 2D 81 03 75 26 56 0A 99 7B 20  ......-..u&V..{
00000040: 65 19 74 5C DD 21                                e.t\.!
pi@raspberrypi:~/auth_demo $
```

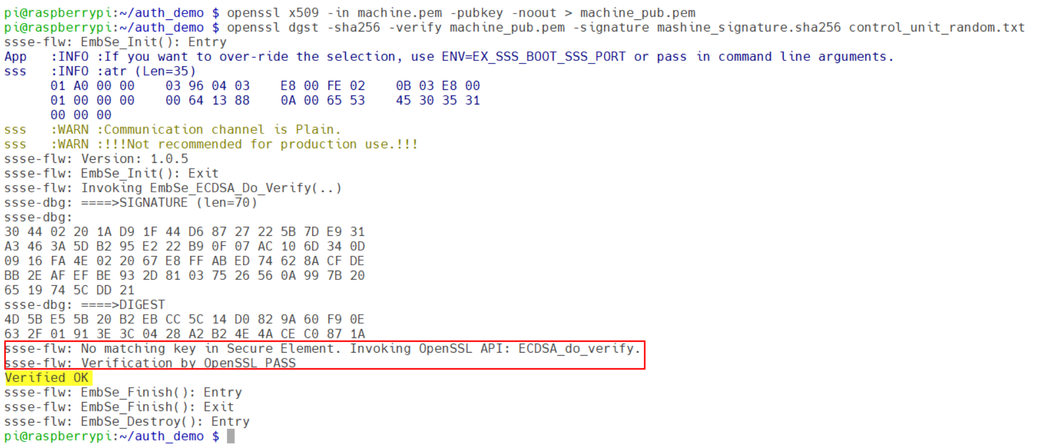
Figure 42. Machine signature

The control unit extracts the unique machine public key from certificate using the following OpenSSL command:

```
openssl x509 -in machine.pem -pubkey -noout > machine_pub.pem
```

Finally, the control unit verifies the signature with the machine public key `machine_pub.pem`.

```
openssl dgst -sha256 -verify machine_pub.pem -signature
mashine_signature.sha256 control_unit_random.txt
```



```
pi@raspberrypi:~/auth_demo $ openssl x509 -in machine.pem -pubkey -noout > machine_pub.pem
pi@raspberrypi:~/auth_demo $ openssl dgst -sha256 -verify machine_pub.pem -signature mashine_signature.sha256 control_unit_random.txt
ssse-flw: EmbSe_Init(): Entry
App    :INFO :If you want to over-ride the selection, use ENV=EX_SSS_BOOT_SSS_PORT or pass in command line arguments.
sss    :INFO :atr (Len=35)
      01 A0 00 00    03 96 04 03    E8 00 FE 02    0B 03 E8 00
      01 00 00 00    00 64 13 88    0A 00 65 53    45 30 35 31
      00 00 00
sss    :WARN :Communication channel is Plain.
sss    :WARN :!!!Not recommended for production use.!!!
ssse-flw: Version: 1.0.5
ssse-flw: EmbSe_Init(): Exit
ssse-flw: Invoking EmbSe_ECDSA_Do_Verify(..)
ssse-dbg: ====>SIGNATURE (len=70)
ssse-dbg:
30 44 02 20 1A D9 1F 44 D6 87 27 22 5B 7D E9 31
A3 46 3A 5D B2 95 E2 22 B9 0F 07 AC 10 6D 34 0D
09 16 FA 4E 02 20 67 E8 FF AB ED 74 62 8A CF DE
BB 2E AF EF BE 93 2D 81 03 75 26 56 0A 99 7B 20
65 19 74 5C DD 21
ssse-dbg: ====>DIGEST
4D 5B E5 5B 20 B2 EB CC 5C 14 D0 82 9A 60 F9 0E
63 2F 01 91 3E 3C 04 28 A2 B2 4E 4A CE C0 87 1A
ssse-flw: No matching key in Secure Element. Invoking OpenSSL API: ECDSA_do_verify.
ssse-flw: Verification by OpenSSL PASS
Verified OK
ssse-flw: EmbSe_Finish(): Entry
ssse-flw: EmbSe_Finish(): Exit
ssse-flw: EmbSe_Destroy(): Entry
pi@raspberrypi:~/auth_demo $
```

Figure 43. OpenSSL - Verify machine signature

The machine is authenticated in case OpenSSL returns `Verified OK`.

4.8 Binding A5000 to a host MCU/MPU using Platform SCP

Binding is a process to establish a pairing between the IoT device host MPU/MCU and A5000, so that only the paired MPU/MCU is able to use the services offered by the corresponding A5000 and vise versa.

A mutually authenticated, encrypted channel will ensure that both parties are indeed communicating with the intended recipients and that local communication is protected against local attacks, including man-in-the-middle attacks aimed at intercepting the communication between the MPU/MCU and the A5000 and physical tampering attacks aimed at replacing the host MPU/MCU or A5000 .

A5000 natively supports Global Platform Secure Channel Protocol 03 (SCP03) for this purpose. PlatformSCP uses SCP03 and can be enabled to be mandatory.

This chapter describes the required steps to enable Platform SCP in the middlware for A5000.

The following topics are discussed:

- Section 4.8.1Introduction to the Global Platform Secure Channel Protocol 03 (SCP03)
- Section 4.8.2 How to enable Platform SCP in the Plug & Trust Middleware
- Section 4.8.3 How to configure the A5000 product specific SCP keys in the Plug & Trust Middleware

4.8.1 Introduction to the Global Platform Secure Channel Protocol 03 (SCP03)

The Secure Channel Protocol SCP03 authenticates and protects locally the bidirectional communication between host and A5000 against eavesdropping on the physical I2C interface.

A5000 can be bound to the host by injecting in both the host and A5000 the same unique SCP03 AES key-set and by enabling the Platform SCP feature in the Plug & Trust Middleware. The AN12662 *Binding a host device to EdgeLock SE05x* describes in detail the concept of secure binding.

SCP03 is defined in Global Platform Secure Channel Protocol '03' - Amendment D v1.2 specification.

SCP03 can provide the following three security goals:

- **Mutual authentication (MA)**
  - Mutual authentication is achieved through the process of initiating a Secure Channel and provides assurance to both the host and the A5000 entity that they are communicating with an authenticated entity.
- **Message Integrity**
  - The Command- and Response-MAC are generated by applying the CMAC according to NIST SP 800-38B.
- **Confidentiality**
  - The message data field is encrypted across the entire data field of the command message to be transmitted to the A5000, and across the response transmitted from the A5000.

The SCP03 secure channel is set up via the A5000 authenticator application using the standard ISO7816-4 secure channel APDUs.

The establishment of an SCP03 channel requires three static 128-bit AES keys shared between the two communicating parties: `Key-ENC`, `Key-MAC` and `Key-DEK`.

`Key-ENC` and `Key-MAC` keys are used during the SCP03 channel establishment to generate the session keys. Session Keys are generated to ensure that a different set of keys are used for each Secure Channel Session to prevent replay attacks.

`Key-ENC` is used to derive the session key `S-ENC`. The `S-ENC` key is used for encryption/decryption of the exchanged data. The session keys `S-MAC` and `R-MAC` are derived from `Key-MAC` and used to generate/verify the integrity of the exchanged data (C-APDU and R-APDU).

`Key-DEK` key is used to encrypt new SCP03 keys in case they get updated.

Table 2. Static SCP03 keys

Key	Description	Usage	Key Type
`Key-ENC`	Static Secure Channel Encryption Key	Generate session key for Decryption/ Encryption (AES)	AES 128
`Key-MAC`	Static Secure Channel Message Authentication Code Key	Generate session key for Secure Channel authentication and Secure Channel MAC Verification/Generation (AES)	AES 128
`Key-DEK`	Data Encryption Key	Sensitive Data Decryption (AES)	AES 128

The session key generation is performed by the Plug & Trust Middleware host crypto.

Table 3. SCP03 session keys

Key	Description	Usage	Key Type
`S-ENC`	Session Secure Channel Encryption Key	Used for data confidentiality	AES 128
`S-MAC`	Secure Channel Message Authentication Code Key for Command	Used for data and protocol integrity	AES 128
`S-RMAC`	Secure Channel Message Authentication Code Key for Response	User for data and protocol integrity	AES 128

Note: *For further details please refer to Global Platform Secure Channel Protocol '03' - Amendment D v1.2.*

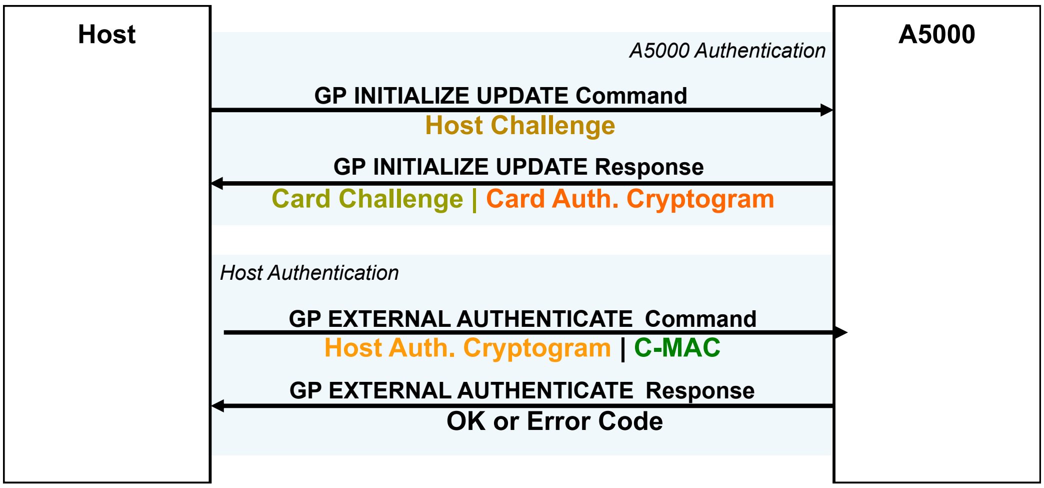


Figure 44. SPC03 mutual authentication – principle

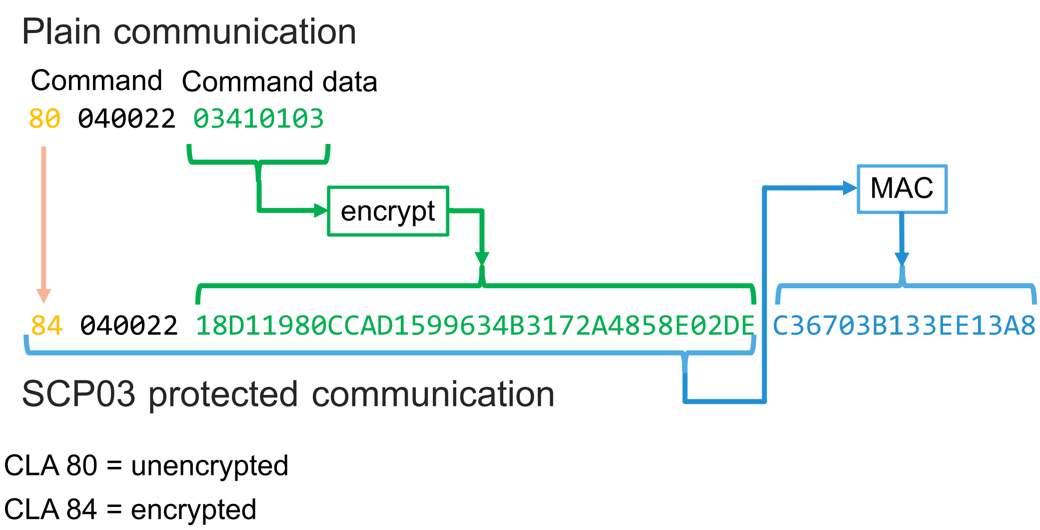


Figure 45. SPC03 Encryption and MACing principle

4.8.2 How to enable Platform SCP in the Plug & Trust Middleware

To enable Platform SCP it is required to rebuild the Plug & Trust Middleware with the following CMake setting:

- Select ***SCP03_SSS*** for the CMake option ***PTWM_SCP***.
- Select ***PlatfSCP03*** for the CMake option ***PTWM_SE05X_Auth***.

The project settings can be specified dynamically using the CMake GUI. The figure below shows a CMake GUI screenshot with the required project settings.

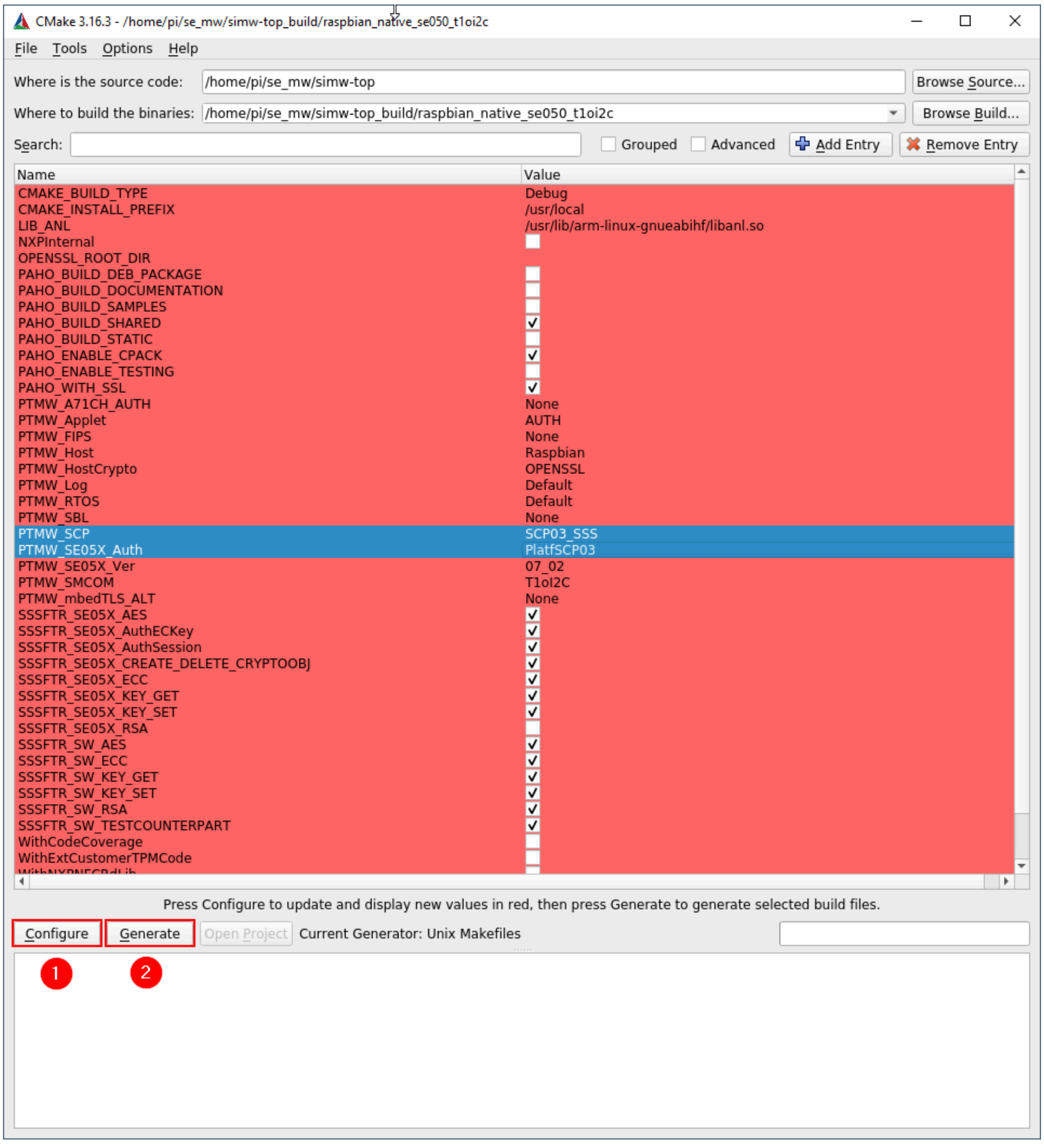


Figure 46. A5000 CMake options to enable Platform SCP

Run the following commands to update the CMake settings and rebuild the Plug & Trust Middleware:

```
cd ~/se_mw/simw-top_build/raspbian_native_se050_t1oi2c
```



```
cmake-gui .
```

Update the CMake settings as explained above. Press first the ***Configure*** button and second the ***Generate*** button and close the CMake GUI.

```
cmake --build .
```

```
sudo make install
```

```
sudo ldconfig /usr/local/lib/
```

Note: *The* [AN12570 "Quick start guide with Raspberry Pi"](#) *describes how to build the Plug & Trust Middleware in detail (see chapter 3.3. Build EdgeLock SE Plug & Trust Middleware test examples).*

4.8.3 How to configure the A5000 product specific SCP keys in the Plug & Trust Middleware

A5000 is delivered with the default A5000 Platform SCP keys as shown in the table below.

Table 4. 128-bit AES Default Platform SCP keys

Configuration	ENC (hex)	MAC (hex)	DEK (hex)
A5000R	c9118500b5ffa143 3a50226f489a0aa5	29d2fe28f7feeb15 3068be381f61bc01	6124d38402118060 ed910360fc5a4278

By default the Plug & Trust Middleware is configured with default Platform SCP keys for a different product. Therefore, it is required to change the default settings. For evaluation purpose the MW supports to store the Platform SCP key in a plain text file. For further details see Plug & Trust Middleware documentation chapter *11.10 Using own Platform SCP03 keys*.

In this example we use the Linux environment variable `EX_SSS_BOOT_SCP03_PATH` to define the Platform SCP key textfile (filename and location).

The following Linux commands can be used to create the Platform SCP key file (a5000_scp_keys.txt):

```
echo ENC c9118500b5ffa1433a50226f489a0aa5 > a5000_scp_keys.txt
```

```
echo MAC 29d2fe28f7feeb153068be381f61bc01 >> a5000_scp_keys.txt
```

```
echo DEK 6124d38402118060ed910360fc5a4278 >> a5000_scp_keys.txt
```

Check the `a5000_scp_keys.txt` file content:

```
cat a5000_scp_keys.txt
```

```
pi@raspberrypi:~/auth_demo $ echo ENC c9118500b5ffa1433a50226f489a0aa5 > a5000_scp_keys.txt
pi@raspberrypi:~/auth_demo $ echo MAC 29d2fe28f7feeb153068be381f61bc01 >> a5000_scp_keys.txt
pi@raspberrypi:~/auth_demo $ echo DEK 6124d38402118060ed910360fc5a4278 >> a5000_scp_keys.txt
pi@raspberrypi:~/auth_demo $ cat a5000_scp_keys.txt
ENC c9118500b5ffa1433a50226f489a0aa5
MAC 29d2fe28f7feeb153068be381f61bc01
DEK 6124d38402118060ed910360fc5a4278
pi@raspberrypi:~/auth_demo $
```

Figure 47. A5000Platform SCP plain text key file

Note: *In this example the Raspberry Pi is used for evaluation purpose only. Because different host MCU/MPU platforms are providing different hardware security mechanisms* *to protect keys it is not in the scope of this document to demonstrate how to store the Platform SCP shared binding keys securely. For commercial deployment the secure storage of Platform SCP keys must be adapted accordingly.*

In the next step we can verify if we successfully configured the enviroment to support Platform SCP. For this purpose we use again the OpenSSL command `rand` and delegate the random number generation to A5000.

```
export OPENSSL_CONF=~/se_mw/simw-top/demos/linux/common/
openssl11_sss_se050.cnfexport EX_SSS_BOOT_SCP03_PATH=~/
device_to_device_auth_demo/a5000_scp_keys.txt
```

```
openssl rand -hex 8
```

Different to the examples in the previous chapters the bidirectional communication between host and A5000 is protected with Platform SCP.

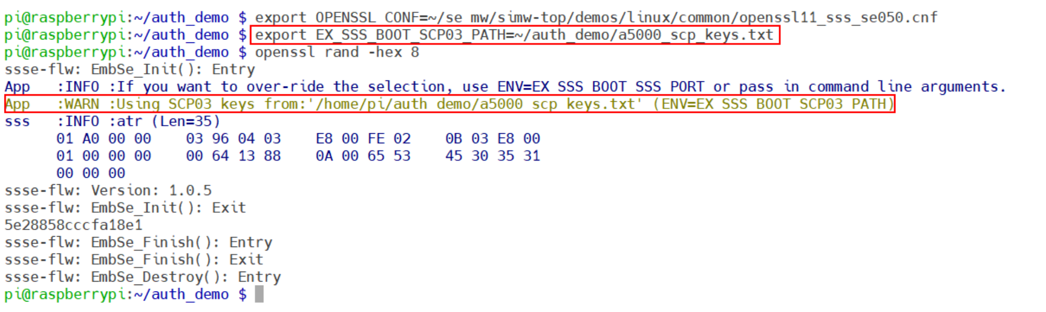


Figure 48. A5000 CMake options to enable Platform SCP

The Plug & Trust Middleware provides the following additional examples to rotate the PlatformSCP Keys and to mandate Platform SCP.

- **SE05X Rotate PlatformSCP Keys example**: Showcases authentication with default Platform SCP keys and the rotation (update) of those keys with user defined keys. The example documentation is available in the EdgeLock SE05x Plug & Trust Middleware documentation (`simw-top/doc/demos/se05x/se05x_RotatePlatformSCP03Keys/Readme.html`). The example source code is available at `/simw-top/demos/se05x/se05x_RotatePlatformSCP03Keys`.
- **SE05X Mandate SCP example**: Showcases how to make Platform SCP authentication mandatory in EdgeLock SE05x. The example documentation is available in the EdgeLock SE05x Plug & Trust Middleware documentation (`/simw-top/doc/demos/se05x/se05x_MandatePlatformSCP/Readme.html`). The example source code is available at `/simw-top/demos/se05x/se05x_MandatePlatformSCP`.
- **SE05x AllowWithout PlatformSCP example**: This project demonstrates how to configure SE05X to allow without platform SCP. The example documentation is available in the EdgeLock SE05x Plug & Trust Middleware documentation (`~/se_mw/simw-top/doc/demos/se05x/se05x_AllowWithoutPlatformSCP/Readme.html`). The example source code is available at `~/se_mw/simw-top/demos/se05x/se05x_AllowWithoutPlatformSCP`.

4.9 Manage access from multiple Linux processes to the A5000

The Plug & Trust Middleware provides the Access Manager to support concurrent access from multiple linux processes to the A5000 authenticator application. The Access Manager can establish a connection to the A5000 authenticator application either as a plain connection or using Platform SCP.

Client processes are connecting over the JRCPv1 protocol to the Access Manager.

Please refer to the Plug & Trust Middleware documentation chapter 5.4.3. Access Manager for more details.

5 A5000 secure provisioning

The IoT device identity should be unique, verifiable and trustworthy so that device registration attempts and any data uploaded to the OEM's servers can be trusted.

The A5000 is designed to provide a tamper-resistant platform to safely store keys and credentials needed for device authentication and registration to OEM's cloud service. Leveraging the A5000 security IC, OEMs can safely authenticate their devices without writing security code or exposing credentials or keys.

The following options are available for provisioning the EdgeLock A5000 security IC:

• **EdgeLock 2GO Ready**: Every EdgeLock A5000 product variant comes pre-provisioned with keys which can be used for all major use cases, including device-to-device authentication.

• **EdgeLock 2GO Custom**: NXP offers a customization service for injecting the credentials that you need during the A5000 IC manufacturing. Please contact NXP for more information on this service.

• **EdgeLock 2GO Managed:** NXP offers a cloud service for remotely configurating your A5000. EdgeLock 2GO Managed is a secure and flexible way for provisioning the keys and certificates required on your devices and to manage the lifecycle of your device credentials.

You can find more information and request an evaluation account at www.nxp.com/EdgeLock2GO .

• **EdgeLock SE05x provisioning by OEMs, distributors or third-party partners**: OEMs can provision EdgeLock A5000 on their own or select a distributor or third-party partner for provisioning the A5000 .

6 References

- DS6676xx, A5000 EdgeLock Secure Authenticator Product data sheet. Available under: https://www.nxp.com/docs/en/data-sheet/A5000-DATASHEET.pdf
- AN12570, EdgeLock SE05x Quick start guide with Raspberry Pi. Available under: https://www.nxp.com/docs/en/application-note/AN12570.pdf

7 Legal information

7.1 Definitions

Draft — A draft status on a document indicates that the content is still under internal review and subject to formal approval, which may result in modifications or additions. NXP Semiconductors does not give any representations or warranties as to the accuracy or completeness of information included in a draft version of a document and shall have no liability for the consequences of use of such information.

7.2 Disclaimers

Limited warranty and liability — Information in this document is believed to be accurate and reliable. However, NXP Semiconductors does not give any representations or warranties, expressed or implied, as to the accuracy or completeness of such information and shall have no liability for the consequences of use of such information. NXP Semiconductors takes no responsibility for the content in this document if provided by an information source outside of NXP Semiconductors. In no event shall NXP Semiconductors be liable for any indirect, incidental, punitive, special or consequential damages (including - without limitation - lost profits, lost savings, business interruption, costs related to the removal or replacement of any products or rework charges) whether or not such damages are based on tort (including negligence), warranty, breach of contract or any other legal theory. Notwithstanding any damages that customer might incur for any reason whatsoever, NXP Semiconductors' aggregate and cumulative liability towards customer for the products described herein shall be limited in accordance with the Terms and conditions of commercial sale of NXP Semiconductors.

Right to make changes — NXP Semiconductors reserves the right to make changes to information published in this document, including without limitation specifications and product descriptions, at any time and without notice. This document supersedes and replaces all information supplied prior to the publication hereof.

Suitability for use — NXP Semiconductors products are not designed, authorized or warranted to be suitable for use in life support, life-critical or safety-critical systems or equipment, nor in applications where failure or malfunction of an NXP Semiconductors product can reasonably be expected to result in personal injury, death or severe property or environmental damage. NXP Semiconductors and its suppliers accept no liability for inclusion and/or use of NXP Semiconductors products in such equipment or applications and therefore such inclusion and/or use is at the customer's own risk.

Applications — Applications that are described herein for any of these products are for illustrative purposes only. NXP Semiconductors makes no representation or warranty that such applications will be suitable for the specified use without further testing or modification. Customers are responsible for the design and operation of their applications and products using NXP Semiconductors products, and NXP Semiconductors accepts no liability for any assistance with applications or customer product design. It is customer's sole responsibility to determine whether the NXP Semiconductors product is suitable and fit for the customer's applications and products planned, as well as for the planned application and use of customer's third party customer(s). Customers should provide appropriate design and operating safeguards to minimize the risks associated with their applications and products. NXP Semiconductors does not accept any liability related to any default, damage, costs or problem which is based on any weakness or default in the customer's applications or products, or the application or use by customer's third party customer(s). Customer is responsible for doing all necessary testing for the customer's applications and products using NXP Semiconductors products in order to avoid a default of the applications and the products or of the application or use by customer's third party customer(s). NXP does not accept any liability in this respect.

Terms and conditions of commercial sale — NXP Semiconductors products are sold subject to the general terms and conditions of commercial sale, as published at http://www.nxp.com/profile/terms, unless otherwise agreed in a valid written individual agreement. In case an individual agreement is concluded only the terms and conditions of the respective agreement shall apply. NXP Semiconductors hereby expressly objects to applying the customer's general terms and conditions with regard to the purchase of NXP Semiconductors products by customer.

Export control — This document as well as the item(s) described herein may be subject to export control regulations. Export might require a prior authorization from competent authorities.

Evaluation products — This product is provided on an "as is" and "with all faults" basis for evaluation purposes only. NXP Semiconductors, its affiliates and their suppliers expressly disclaim all warranties, whether express, implied or statutory, including but not limited to the implied warranties of non-infringement, merchantability and fitness for a particular purpose. The entire risk as to the quality, or arising out of the use or performance, of this product remains with customer. In no event shall NXP Semiconductors, its affiliates or their suppliers be liable to customer for any special, indirect, consequential, punitive or incidental damages (including without limitation damages for loss of business, business interruption, loss of use, loss of data or information, and the like) arising out the use of or inability to use the product, whether or not based on tort (including negligence), strict liability, breach of contract, breach of warranty or any other theory, even if advised of the possibility of such damages. Notwithstanding any damages that customer might incur for any reason whatsoever (including without limitation, all damages referenced above and all direct or general damages), the entire liability of NXP Semiconductors, its affiliates and their suppliers and customer's exclusive remedy for all of the foregoing shall be limited to actual damages incurred by customer based on reasonable reliance up to the greater of the amount actually paid by customer for the product or five dollars (US$5.00). The foregoing limitations, exclusions and disclaimers shall apply to the maximum extent permitted by applicable law, even if any remedy fails of its essential purpose.

Translations — A non-English (translated) version of a document is for reference only. The English version shall prevail in case of any discrepancy between the translated and English versions.

Security — Customer understands that all NXP products may be subject to unidentified or documented vulnerabilities. Customer is responsible for the design and operation of its applications and products throughout their lifecycles to reduce the effect of these vulnerabilities on customer's applications and products. Customer's responsibility also extends to other open and/or proprietary technologies supported by NXP products for use in customer's applications. NXP accepts no liability for any vulnerability. Customer should regularly check security updates from NXP and follow up appropriately. Customer shall select products with security features that best meet rules, regulations, and standards of the intended application and make the ultimate design decisions regarding its products and is solely responsible for compliance with all legal, regulatory, and security related requirements concerning its products, regardless of any information or support that may be provided by NXP. NXP has a Product Security Incident Response Team (PSIRT) (reachable at PSIRT@nxp.com) that manages the investigation, reporting, and solution release to security vulnerabilities of NXP products.

7.3 Trademarks

Notice: All referenced brands, product names, service names and trademarks are the property of their respective owners.

NXP — wordmark and logo are trademarks of NXP B.V.

EdgeLock — is a trademark of NXP B.V.

Tables

Figures

Contents

Date of release: 14 September 2022
Document identifier: AN13500