Security Tool

This document describes the security features used in RA6W1 and RA6W2, as well as in the corresponding modules RRQ61001 and RRQ61051.

Contents

Figures

Tables

1. Terms and Definitions

AHB	AMBA Advanced High-performance Bus
CM	Chip Manufacture
CMPU	Chip Master Process Unit
DCU	Debug Control Unit
DM	Device Manufacture
DMPU	Device Master Process Unit
HUK	Hardware Unique Key
IVT	Interrupt Vector Table
LE	Little Endian
LCS	Lifecycle States
OEM	Original Equipment Manufacturer
OS	Operating System
OTP	One-Time Programmable
PoR	Power-On Reset
RMA	Return Merchandise Authorization
RoT	Root of Trust
SB	Secure Boot
SD	Secure Debug

2. RA6W1/RA6W2 Security

2.1 Security Engine

RA6W1 and RA6W2 use the Arm® CryptoCell-312 as its security engine that provides security services for platforms such as Secure Boot and Key Management with acceleration for cryptographic operations. Many of the security services are implemented in the ROM code, for example, the Secure Boot process. The cryptography and management service are integrated into the Operating System (OS) and are used with MbedTLS for the TLS and SSL protocols.

2.2 Hardware Components

Figure 1 shows a block diagram of the RA6W1/RA6W2 security engine.

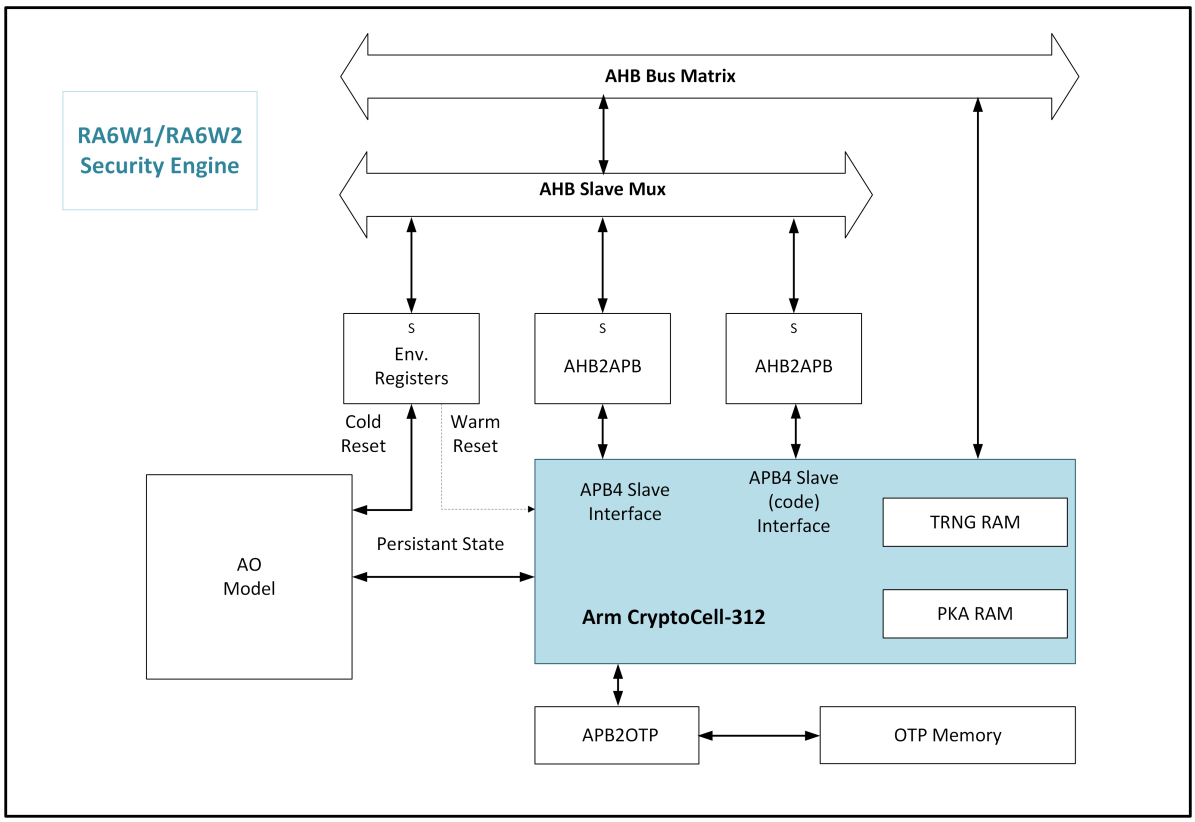

Figure 1. Block diagram of RA6W1/RA6W2 security engine

The host processor can access CC312's SRAM and registers, as well as the One-Time Programmable (OTP) memory in the RA6W1/RA6W2. The CC312 security engine can initialize transactions with the system memory or other DMA slaves through the AMBA Advanced High-performance Bus (AHB) Master.

The CC312 security engine is connected to the external OTP memory through the Advanced Peripheral Bus (APB4) Master interface, and OTP memory holds the device root key (HUK) and lifecycle state (LCS).The specific area of OTP memory that is controlled by the CC312 security engine is only accessible by the CC312, and therefore, acts as the Root of Trust for RA6W1 and RA6W2.

The Always-On (AO) module must survive a power-down of the CC312 to keep the critical state of the embedded system. The AO module includes the following components:

- Security Lifecycle States
- Debug Control Unit (DCU) and DCU Lock Registers
- Lock-Bits Register

2.3 Software Architecture

Secure Boot services run from the ROM in the RA6W1/RA6W2. The crypto services, which are accelerated by CC312 hardware, can be used with mbedTLS APIs. See the open-source Mbed TLS APIs documentation for API details. Figure 2 shows the security software architecture in the RA6W1/RA6W2.

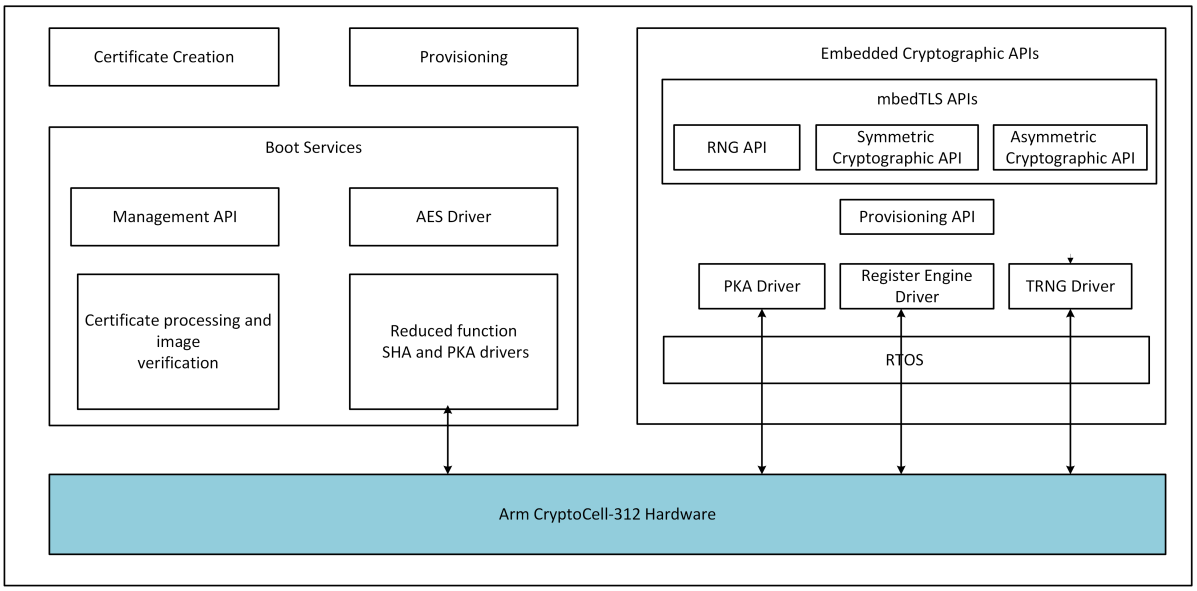

Figure 2. RA6W1/RA6W2 security software architecture

3. Security Features

3.1 Security Services

3.1.1 Secure Boot

The RA6W1/RA6W2 provides a Secure Boot function that allows trusted images signed with a key matching the registration information in the system during the boot process to ensure the system's platform integrity. In the production step, it is necessary to register the key information for authentication in the OTP memory, which is protected by CC312.

3.1.2 Secure Debug

The RA6W1/RA6W2 supports a Secure Debug function that provides hardware protection of the debug port to prevent an external security attack. When you need to enable this port for system debugging, Secure Debug uses the authenticated key with the signed debug certificate to remove the hardware protection, to allow debugging tasks.

3.1.3 Secure Asset

Secure Asset is a cryptographic service provided to protect data stored in external storage (Serial Flash memory). Data can be encrypted or decrypted with the provisioning key stored in the chip. Production-Line Provisioning is used to protect the data used in the mass production process, and Asset Provisioning is used to protect the data used during system operation.

3.2 Security Keys

This section describes the required security keys in the RA6W1/RA6W2. For the security feature in the RA6W1/RA6W2, several security keys are required and should be stored in OTP memory before production. All secret keys are burned using the Security Tool. All hardware keys are accessed only by the Arm CryptoCell-312 and cannot be read by the CPU depending on the security LCS.

3.2.1 HUK

Hardware Unique Key (HUK) is called the device key and a secret value that is burned into OTP memory and is read by hardware as part of the secure boot sequence and is no longer accessible for reading. HUK can only be used by the AES engine, and only for the derivation of other keys. It must be unique per device. For this uniqueness, it is generated as the seed value derived from TRNG in CC312. SoC-ID is derived from this key. A SoC-ID is required in Secure Debug and only valid in the Secure LCS state. HUK is generated by Security Tool.

3.2.2 Platform Key

The Platform key (Krtl) is placed in the RA6W1/RA6W2 and used for provisioning during the production lifecycle (CM and DM LCS). The Platform key should be provided by Renesas Electronics when requested. It is a 128-bit AES class key and a random 128-bit value. A key derived from this key is used to encrypt the provisioning assets such as Chip Manufacturer keys and Device Manufacturer keys, which are described in the following section. This key is only for use in Chip Manufacturer (CM) and Device Manufacturer (DM) LCS and is locked by hardware in all other LCS. Krtl should not be exposed to others for any reason. The Security Tool uses this key in Secure Production and removes the key after use.

3.2.3 Chip Manufacturer Key

CM keys are burned in OTP memory at production time and used as a back-up key for DM keys. The CM keys are generated in the Security Tool. There are two types of CM keys:

- CM Provisioning Key (Kpicv): A 128-bit AES key used for asset provisioning flow.
- CM Encryption Key (Kceicv): A 128-bit AES key used to encrypt or decrypt software images as part of the Secure Boot process.

3.2.4 Device Manufacturer Key

DM keys are burned in OTP memory at production time. They are generated in the Security Tool. There are two types of DM keys:

- DM Provisioning Key (Kcp): A 128-bit AES key used for asset provisioning.
- DM Encryption Key (Kce): A 128-bit AES key used to encrypt or decrypt software images as part of the Secure Boot process.

3.2.5 RoT

The Root of Trust (RoT) is a hash of the public key. Every public key has a corresponding private key that must be preserved and not exposed for security reasons. These public and private key pairs are generated in the Security Tool. There are two RoT keys: Hbk0 and Hbk1. Hbk0 is a hash of the CM public key generated by the Security Tool and is a backup RoT for Hbk1 (a hash of the DM public key), which is normally used for Secure Boot and Secure Debug. Both Hbk0 and Hbk1 should be burned to the OTP memory as RoT. Hbk0 and Hbk1 are used to validate the authentication of an image with certificate data.

Summary of Hbk0 and Hbk1

- Hbk0
  A 128-bit truncated SHA-256 digest derived from a CM public key. It serves as a backup key for Hbk1 and is primarily used for Secure Boot and Secure Debug operations.
- Hbk1
  A 128-bit truncated SHA-256 digest of a DM public key. Used as a main RoT key.

3.3 OTP Memory

OTP memory is used to store keys and configure data. The RA6W1/RA6W2 has 2 kB of OTP memory. Mandatory configuration data must be burned at the offsets given in Table 1.

Table 1. Configuration data and key in OTP memory

32-bit Word (Note 1)	Description	Read	Write
0x00-0x07	HUK	Readable only in CM LCS	Writable only in CM or RMA LCS
0x0B	Kpicv	Readable only in CM LCS	Writable only in CM or RMA LCS
0x0C-0x0F	Kceicv	Readable only in CM LCS	Writable only in CM or RMA LCS
0x10	CM programmed flags	See Table 2	Writable only in CM or RMA LCS
0x11-0x18	RoT public key If split into CM and DM keys: ■ CM key (Hbk0): 0x11-0x14 ■ DM key (Hbk1): 0x15-0x18	Readable in all LCS	Writable in CM or DM LCS
0x19-0x1C	Kcp	Readable in CM LCS or DM LCS	Writable in DM or RMA LCS
0x1D-0x20	Kce	Readable in CM or DM LCS	Writable in DM or RMA LCS
0x21	DM programmed flags	See Table 3	Writable in all LCS
0x27	General purpose configuration flags		Writable only in CM LCS
0x28-0x2B	DCU 128 bits lock mask that allows software to lock the required debug bits	Readable in all LCS	Writable in CM or DM LCS
0x40-0x1FE	Code and data sections that you can use	Readable in all LCS	

Note 1 The word area from 0x00 ~ 0x2B is not accessible by the CPU and is accessible only by the hardware security engine in the RA6W1/RA6W2.

The word area from 0x00 ~ 0x2B should be burned into OTP memory at production time. For this purpose, special binary images called CMPU and DMPU are required. CMPU is a binary image containing the HUK, Hbk0, and CM keys. DMPU is a binary image containing the Hbk1 and DM keys. The CMPU and DMPU binary images are generated by the Security Tool during the Secure Production process. See Section 4.4 Secure Production.
Table 2 shows the CM programmed flags that are located at address 0x10 in the OTP memory.

Table 2. CM programmed flags

Bits	Usage	Read access	Write access
[7:0]	Number of zero bits in HUK	Readable only in CM LCS; masked for reading in any other LCS	Writable in CM LCS and RMA LCS
[14:8]	Number of zero bits in Kpicv (128-bit)	Readable only in CM LCS	Writable in CM LCS and RMA LCS
[15]	Kpicv "not in use" bit	Readable in all security lifecycle states	Writable in CM LCS and RMA LCS
[22:16]	Number of zero bits in Kceicv	Readable only in CM LCS	Writable in CM LCS and RMA LCS
[23]	Kceicv "not in use" bit. If Kceicv is not in use, this bit should be set by the IFT	Readable in all security lifecycle states	Writeable in CM LCS and RMA LCS
[30:24]	Number of zero bits in Hbk0	Readable in all security lifecycle states	Writable in CM LCS and RMA LCS
[31]	Hbk0 "not in use" bit. If Hbk0 is not in use, this bit should be set by the IFT	Readable in all security lifecycle states	Writable in CM LCS and RMA LCS

Table 3 shows the DM programmed flags that are located at address 0x21 in OTP memory.

Table 3. DM programmed flags

Bits	Usage	Read Access	Write Access
[7:0]	Number of zero bits in Hbk1 or Hbk	Readable in all security lifecycle states	Writable in DM LCS and RMA LCS
[14:8]	Number of zero bits in Kcp (128-bit)	Readable only in CM LCS and DM LCS	Writable in DM LCS and RMA LCS
[15]	Kcp "not in use" bit	Readable in all security lifecycle states	Writable in DM LCS and RMA LCS
[22:16]	Number of zero bits in Kce	Readable only in CM LCS and DM LCS	Writable in DM LCS and RMA LCS
[23:23]	Kce "not in use" bit. If Kce is not in use, this bit should be set by the OFT	Readable in all security lifecycle states	Writable in DM LCS and RMA LCS
[29:24]	Reserved	Always readable	Always writable
[30]	DM RMA LCS flag	Readable in all security lifecycle states	Writable in CM LCS, DM LCS and Secure LCS
[31]	CM RMA LCS flag	Readable in all security lifecycle states	Writable in CM LCS, DM LCS, and Secure LCS, only if the CM RMA locking bit in the AO module is not set

3.4 Lifecycle States

Lifecycle states have CM, DM, Secure, and RMA. Figure 3 shows the transitions of Lifecycle States (LCS) of the RA6W1/RA6W2. This enables the device to behave differently in each LCS, protecting any security assets when they are introduced into the device and reducing the risk of IP theft and reverse engineering.

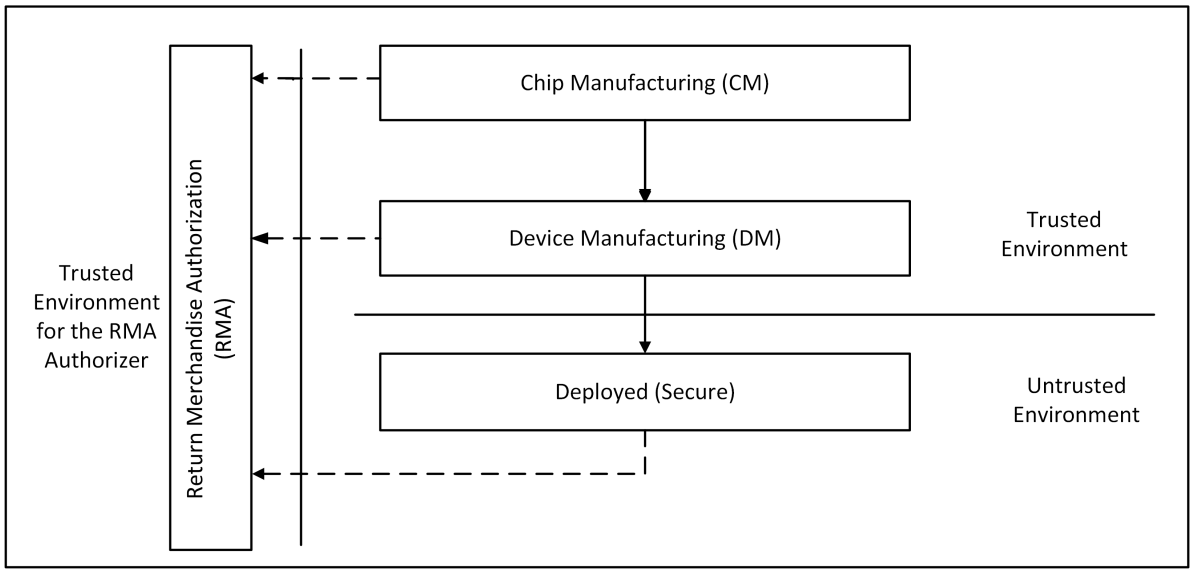

Figure 3. LCS transitions

3.4.1 CM LCS

The device is in CM LCS if the following is true:

- CM-programmed flags: OTP word 0x10 = 0
- DM-programmed flags: OTP word 0x21 = 0

Therefore, the default hardware state is CM LCS. In this LCS, all debug interfaces (UART and JTAG) are enabled.
A CMPU package binary image that is generated with the Security Tool includes the following assets and should be burned into OTP in CM LCS:

- HUK: OTP word 0x00-0x07
  The number of zero bits in HUK: Bits[7:0] of OTP word 0x10
- Hbk0: OTP word 0x11-0x14
  The number of zero bits in Hbk0: Bits[30:24] of OTP word 0x10
  General purpose configuration (GPPC) flags, OTP word 0x27
- CM DCU locking if Hbk0 is used

When these assets are burned, the device does a power-on reset (PoR) to transition to DM LCS.

3.4.2 DM LCS

The device is in DM LCS if the following is true:

- CM-programmed flags: OTP word 0x10 ≠ 0
- DM programmed flags: Bits[7:0] of OTP word 0x21 = 0

In this LCS, all debug interfaces (UART and JTAG) are still enabled.
A DMPU package binary image that is generated with the Security Tool includes the following assets and it should be burned into OTP in DM LCS:

- Hbk1: OTP word 0x15-0x18
  The number of zero bits in Hbk1: Bits[7:0] of OTP word 0x21
- Optional: DM DCU locking if Hbk1 is used

When these assets are burned, the device does a PoR to transition to Secure LCS.

3.4.3 Secure LCS

The Deployed (Secure) LCS is used for devices out of the manufacturing line and in the field. It permits the execution of security functions but blocks all debugging and testing capabilities. Use of Secure Boot is mandatory in this LCS. The device is in Secure LCS if the following is true:

- CM programmed flags: OTP word 0x10 ≠ 0
- DM programmed flags: Bits[7:0] of OTP word 0x21 ≠ 0

Secure LCS is the state changed with the DMPU process, see Section 4.2 Secure Key Generation for more information. This is the state that should be applied at mass production when secure boot is required. When in this state, the debug interface such as JTAG cannot be used anymore for security reasons. To enable the disabled debug interface, a firmware image with a Debug certificate must be used as described in Section 4.5 Create Secure Boot Image.

3.4.4 RMA LCS

The Return Merchandise Authorization (RMA) LCS is a terminal state for devices that are returned to a Chip maker (for example, Renesas Electronics) for analysis of fatal failures. When a device is put into RMA LCS, it loses its existing secret keys but regains full access to all debugging and testing capabilities. All cryptographic engines are usable for test purposes, but the root keys change for each boot phase.

- HUK is replaced with a different random value with each boot cycle. Therefore, any previously saved data that is protected by a key derived from HUK is lost.
- Kce and Kceicv are invalidated so that Secure Boot can be used only in non-encrypted mode.
- Kcp and Kpicv are invalidated so that provisioning can no longer be done based on the previous values.

There are two separate certificates needed to enter a device into RMA LCS – CM RMA and DM RMA.

- CM RMA is a certificate image with CM RoT (Hbk0) chain and removes CM keys in OTP.
- DM RMA is a certificate image with DM RoT (Hbk1) chain and removes DM keys.

For detailed information about the process, see Section 4.6 Key Renewal.

3.5 Boot Services

The boot services in the RA6W1/RA6W2 include the Secure Boot and Secure Debug certificate-based mechanisms that use an RSA private-public key scheme. Secure Boot and Secure Debug are based on the following elements:

- OTP secrets
  Provisioned to the device during the device manufacturing stage (CM LCS or DM LCS).
- ROM code
  A code library linked into the ROM of the device.
- RSA scheme verification
  Verification of Secure Boot and Secure Debug is done over a certificate chain that is two or three certificates long. Each certificate is signed and verified with an RSA PSS scheme (RSA 3072 Private Public Key scheme and compliant to PKCS#1 Ver. 2.1, RSA-PSS).

3.5.1 Secure Boot

Secure Boot guarantees that only authenticated software images (optionally encrypted) are loaded on a target system. A certificate is a message used to prove ownership of a public key. The certificate contains information about the public key, the authentication hash of the next key, and the signature that verifies contents. Figure 4 shows the general structure of a certificate.

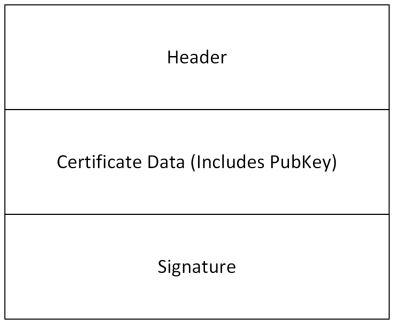

Figure 4. General structure of certificate

A signature is generated by encrypting a hash of the Certificate Data using a private key. Signature verification is done using a public key to decrypt a hash of the same Certificate Data. If the Certificate Data is compromised for any reason, then the decrypted hash of the Certificate Data would be different from the original signature and certificate verification fails.
The RA6W1/RA6W2 uses a Certificate Chain for secure certificate verification.
A three-level "self-signed" certificates chain is used, which is a series of certificates that contain the public keys and are signed with a corresponding private key.
The Secure Boot certificate chain is composed of key certificates and content certificates.

- Key certificates
  Mainly the first or second certificate in the certificates chain.
- Content certificates
  The last certificate in a certificates chain, which is used to load and validate software components.

Figure 5 shows a three-certificate chain.

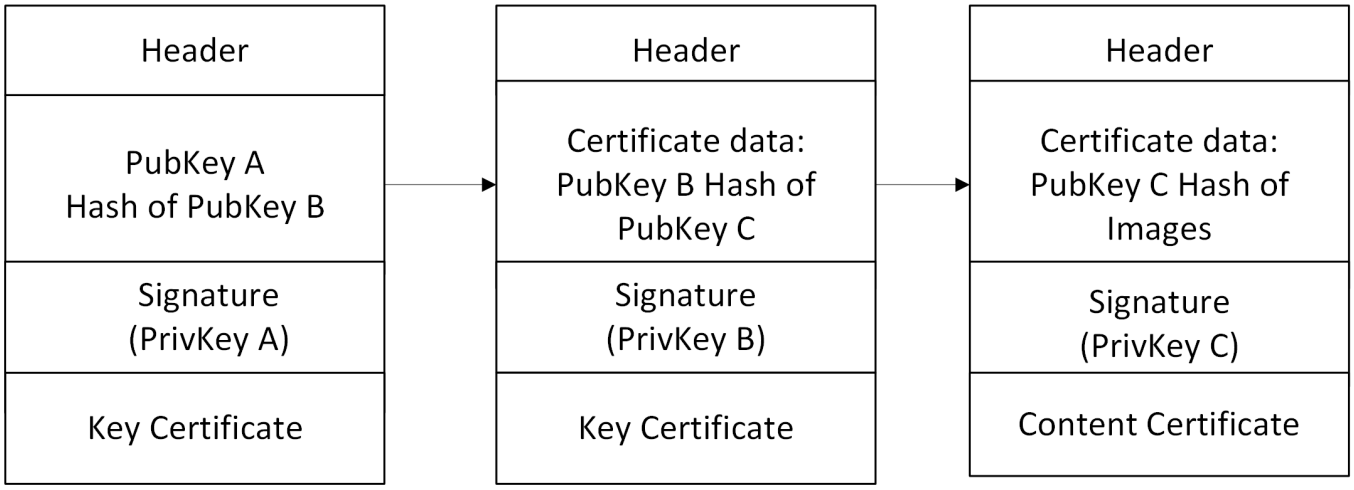

Figure 5. Three-certificate chain

- Three-level SB certificate scheme

The order of three-level SB certificate chain is: master key certificate > key certificate > content certificate.
Even when a key used in a third or second certificate is leaked, it can be replaced with another key if the private key used in the first certificate is not compromised.

3.5.1.1 Secure Boot Flow

To verify a certificate, complete the following steps in the RA6W1/RA6W2:

1. Get the public key from the certificate and calculate its hash (HBK1 or HBK0).
2. Verify the calculated hash:
   - If it is the first certificate in the chain, compare it with the hash value stored in OTP.
   - Otherwise, compare it with the saved hash from the previous certificate in the chain.
3. Verify the RSA signature with the public key of the certificate.
4. Save the public key hash of the next certificate, unless it is the last certificate in the chain.

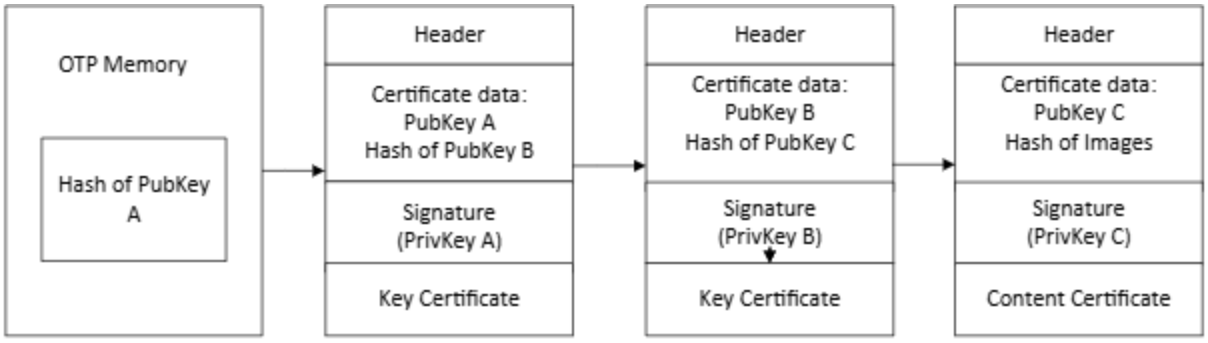

Figure 6. Secure boot flow

The entire certificate chain in Figure 6 is included in the built Image of the RA6W1/RA6W2. And it is impossible for an unauthorized image to boot because of the verification process with this certificates chain.

Figure 7 shows the overall certificate verification process.

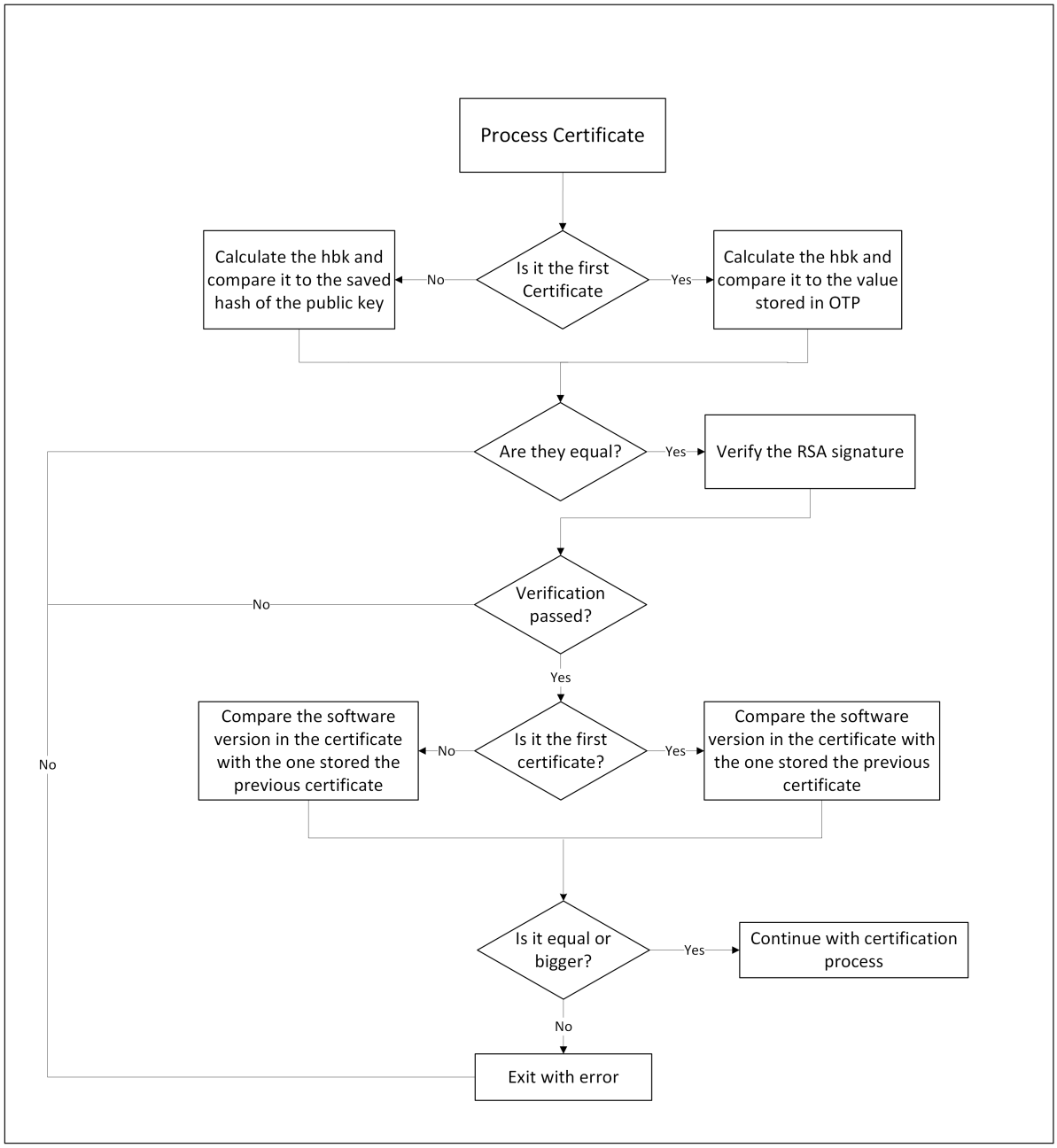

Figure 7. Overall certificate verification process

Figure 8 shows how the content certificate process is done in a loop to process every software image that is signed in the certificate.
The content certificate contains the following information for every image that must be verified:

- The address that the software image is loaded to [load address].
- The flash address that the software image is stored in [storage address].
- The size of the software image.

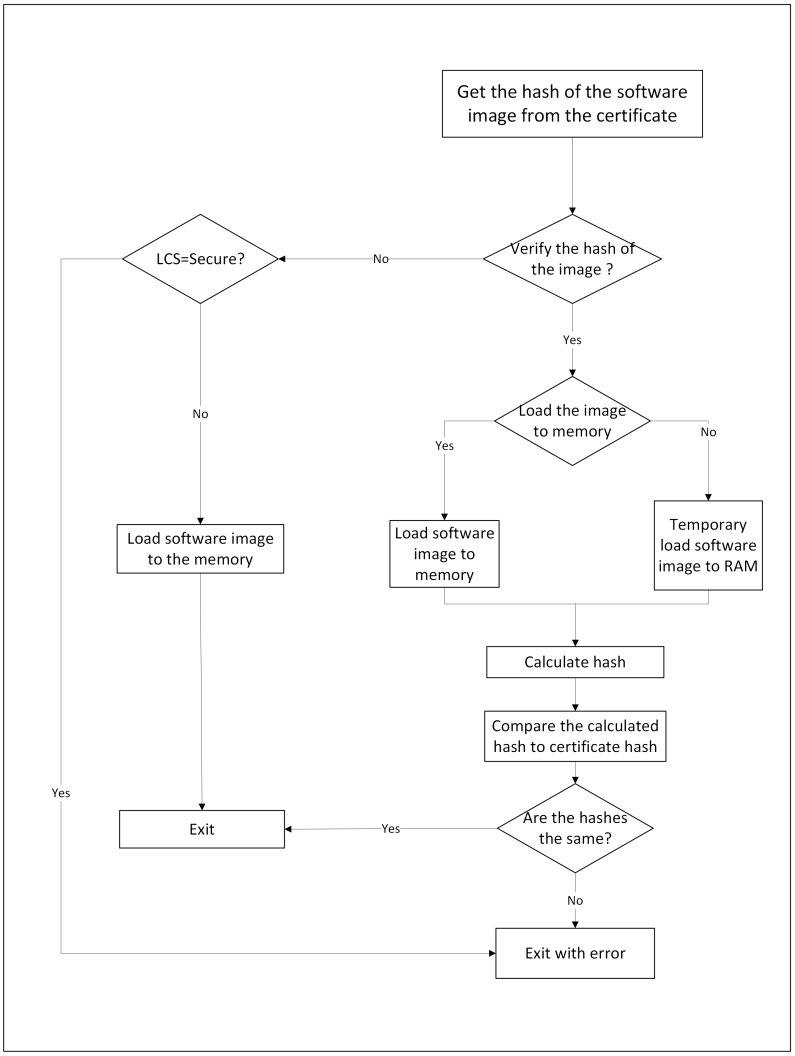

Figure 8. Certification contents in software images

Figure 9 shows the structure of the built image of the RA6W1/RA6W2. The key and content certificate chain are included in the image.
Besides certificates, the following contents are included in the image of the RA6W1/RA6W2:

- Debug certificate (optional)
- Software component (maximum of three components available).

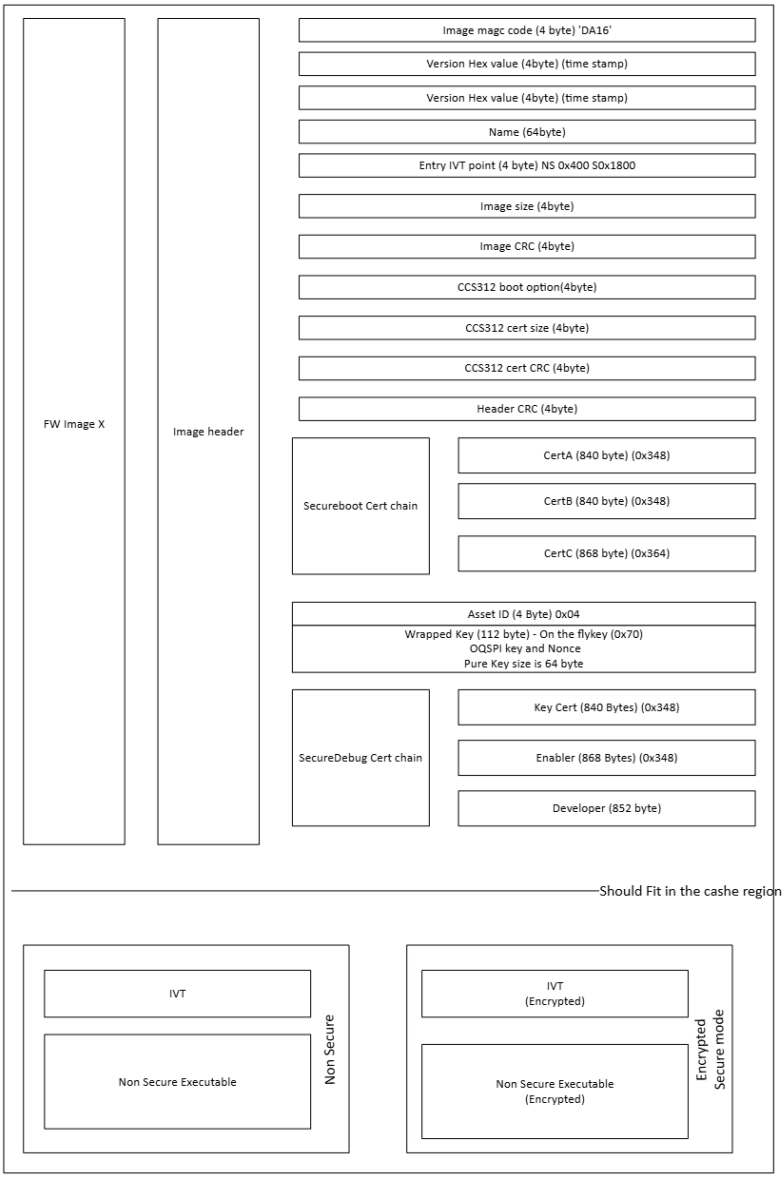

Figure 9. Certification contents in RA6W1/RA6W2

Table 4 shows brief information about the Image header layout.

Table 4. Image header layout

Field	Size (Byte)	Description
Image identifier	4	Image magic code "DA16".
Version HEX value	4 (Little Endian (LE) format)	Usually, it can be filled with a timestamp.
FW name	64	Firmware name.
IVT point	4 (LE format)	Address to start of IVT, relative to the start address of the image. Non-secure 0x400. Secure 0x1800 (Note 1). Flashless secure 0x14D0 (Note 2).
Image size	4 (LE format)	Size of the image.
Image CRC	4 (LE format)	CRC value of the Image. CRC32 is used for it.
CC312 boot option	4	■ NULL – Non-secure booting. ■ "enc" secure booting with wrapped key. ■ "raw" secure Booting without code encrypt. ■ "d" secure booting check debug cert. ■ "n" secure booting without debug cert. For example: ■ "encd" code encrypted and need to check debug cert. ■ "rawd" code is not encrypted and need to check debug cert. ■ "rawn" code is not encrypted and there is no need to check the debug cert. ■ "encn" code is encrypted and no need to check debug cert.
CC312 cert size	4 (LE format)	Length of CERTs.
CC312 cert CRC value	4 (LE format)	CRC value of the CERT. CRC32 is used for it.
Header CRC	4	Header CRC value. From Magic to CC312 Cert CRC value.
Secure boot Cert A	840	Secure boot mandatory.
Secure boot Cert B	840	Secure boot mandatory.
Secure boot Cert C	868	Secure boot mandatory.
Asset ID	4 (LE format)	It is needed if the code is encrypted.
Wrapped key	112	It contains the key for on-the-fly of the OQSPI. This field could be empty when it runs with a raw version of secure image.
Secure debug key cert	840	Secure boot optional.
Secure debug enabler cert	868	Secure boot optional.
Secure debug developer cert	852	Secure boot optional.
Padding data	X	Padding data FF to IVT.
IVT and binary	XX	Binary Encrypted or not.

Note 1 The lowest bits [9:0] used for the on-the-fly region, this region must not be touched. This means that the IVT should be placed at a location of multiple of 0x400. The IVT for the secure image is placed at location 0x1800.

Note 2 The flash-less secure image also uses the same header format. And the IVT location is 0x14d0 (5328) for reducing padding data.

CertA and CertB are key certificates, and CertC is a content certificate as shown in Figure 9. Content can be an RTOS built from SDK.

- RTOS image (XXRTOSXX.img) built from our SDK includes an RTOS binary as a software component (Comp0). All CertA, CertB, and CertC are generated with the Security Tool and attached to each binary (RTOS) to make a bootable image for the RA6W1/RA6W2.
- CertA and CertB are the same for all images while CertC is different for each image.
- CertA with a Hbk1 (DM RoT) chain is generated with the name of `sboot_hbk1_3lvl_key_chain_issuer.bin` in the `dmpublic` directory in the Security Tool.
- CertB with Hbk1 is generated with the name `sboot_hbk1_3lvl_key_chain_publisher.bin` in the `dmpublic` directory.
- CertC is different from each image because CertC contains information of each image such as content and size.
- CertC for the RTOS binary has the name `sboot_hbk1_cache_cert.bin`.

3.5.2 Secure Debug

Secure Debug is a certificate-based mechanism that uses an RSA private-public key scheme. It enables secure debugging of the device.

Secure Debug supports the following operations:

- Perform boot-time verification of debug certificates that enable authenticated debugging of secure domains. The secure domains are controlled by the Debug Control Unit (DCU) registers on the MCU.
- Allow an authorizing party to shift the device into RMA LCS by using the same certificate mechanism. This is called "Secure RMA".

There are two certificate chains in the debug certificate: an enabler certificate and a developer certificate. An enabler debug certificate can enable certain debug interfaces for you to debug a certain device. You can enter a SoC-ID of the target device to extend this to an actual debug certificate.

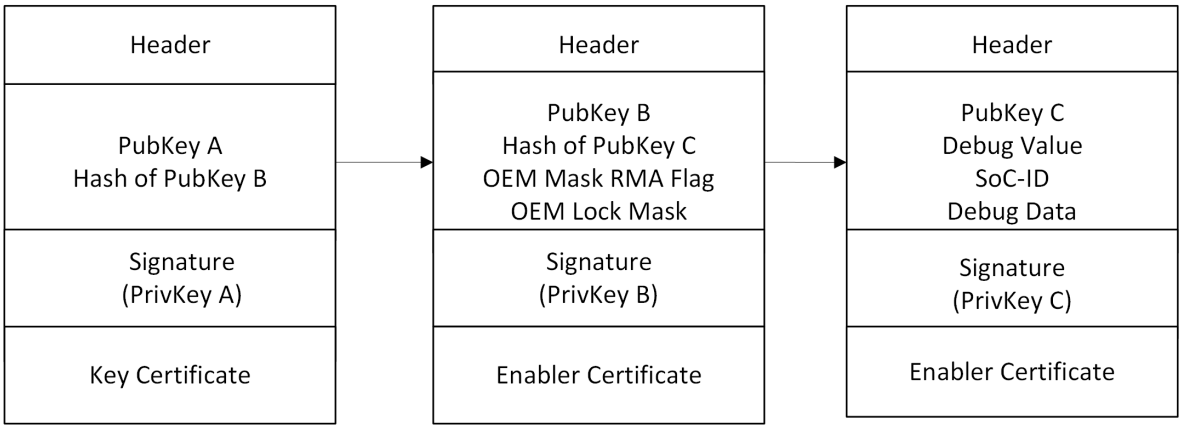

Figure 10. Three-level SD certificate scheme

Developer certificate can be generated with a SoC-ID. If this does not match the target device, then the debug interfaces (JTAG and UART) should not be enabled. When a debug certificate is verified during the boot sequence of the device, the permitted debug interfaces in the DCU mask of the enabler certificate should be enabled (UART0 and JTAG for RA6W1/RA6W2) for the designated device of the SoC-ID in the developer certificate.

An enabler certificate has the following fields:

Table 5. Items in enabler certificate

Items	Condition	Description
RMA-mode	Mandatory if debug-mask is not defined. Cannot be defined together with debug-mask.	Define whether to use this certificate for entry into RMA LCS, by setting it to a non-zero value. Set when "Secure RMA" is running in the Security Tool.
debug-mask	Mandatory if RMA-mode is not defined. Cannot be defined together with RMA-mode.	The DCU mask is allowed by the enabler. A 128-bit mask. Set when "Secure Debug" is running in the Security Tool.
debug-lock	Mandatory if RMA-mode is not defined.	An additional DCU lock mask is required by the enabler and is a 128-bit mask. These bits are added to the OTP-based mask.

A developer certificate has the following fields:

Table 6. Items in developer certificate

Items	Condition	Description
SoC-ID	Mandatory	SoC-ID of the device. Developers can enable debug interfaces of the device with this SoC-ID. If trying to enable the debug interface of the device with a different SoC-ID, it fails.
debug-mask	Mandatory if RMA-mode is not defined. Cannot be defined together with RMA-mode.	The DCU mask is allowed by the developer. A 128-bit mask.

A debug certificate is generated at Secure Debug in the Security Tool, which includes a debug-mask configuration in the enabler certificate and a SoC-ID for the developer certificate. An RMA certificate is generated at Secure RMA in the Security Tool, which includes an RMA-mode configuration in the enabler certificate and a SoC-ID for the developer certificate.

3.6 Device Provisioning

Device provisioning refers to burning secret keys and assets in the OTP memory of a device in a secure manner. The CM keys and assets in Table 7 should be burned in the OTP in CM LCS, and the DM keys and assets in Table 8 should be burned in the OTP in DM LCS.

Table 7. CM keys and assets in CM LCS

Key names or assets	Functions
Kpicv and Kceicv	CM key
Hbk0	Root of Trust
Asset	CM DCU lock bits
Asset	Configuration bits (General Purpose Flag)

A CM Provisioning Utility (CMPU) package binary contains all the above items and is generated when Secure Production is run in the Secure Tool.

Table 8. DM keys and assets in DM LCS

Key names or assets	Functions
Kcp and Kce	DM key
Hbk1	Root Of Trust
Asset	DM DCU lock bits

After the secrets and asset are burned in the OTP memory, LCS automatically changes to Secure LCS, and the JTAG debug interface in the RA6W1/RA6W2 is disabled. To enable the JTAG debug interface again, the debug certificate scheme should be applied. The platform key (Krtl) is required to generate a CMPU and DMPU package binary to encrypt all assets as described in Section 4.4.2 Generate CMPU and DMPU Hex Values.

3.7 Secure Asset

After device provisioning, secret keys in the OTP memory can be used to encrypt or decrypt user data in the Flash memory. It provides APIs and procedures to encrypt and manage data to be stored in Flash with the AES CCM method.

3.7.1 API for Secure Assets

Secure assets are encrypted data stored in Flash. Data decryption is done with the key derived from the provisioning key Kpicv or Kcp that is stored in the OTP memory. Therefore, there is no risk of key disclosure. The Security Tool supports the creation of the secure asset, encrypted with a key derived from the provisioning key. The RA6W1/RA6W2 SDK provides an API to decrypt assets with the key derived from the OTP memory keys by the hardware crypto engine.

The Secure Asset Service uses a CMAC algorithm based on AES encryption and has a file size restriction:

- The valid size of unencrypted data must be a multiple of 16 bytes.
- The maximum size of unencrypted data cannot exceed 512 bytes.
- The maximum size of the secure asset is 560 bytes including the header size.

The decryption API provided by the SDK is FC9K_Secure_Asset().

```
uint32_t DA16X_Secure_Asset(uint32_t Owner,
                                                    uint32_t AssetID,
                                                    uint32_t *InAssetData,
                                                    uint32_t AssetSize,
                                                    uint32_t *OutAssetData,
                                                    uint32_t *OutAssetSize)
```

Where:

- Owner
  Key type number. Use **1** for Kpicv, or **2** for Kcp.
- AssetID
  ID information used in the encryption process.
- InAssetData
  Secure Asset Data. This data must be loaded into SRAM since this function does not access Flash.
- AssetSize
  Size of Secure Asset Data.
- OutAssetData
  Decrypted Asset Data. This data must be allocated to SRFAM since this function does not run a memory allocation.
- OutAssetSize
  Actual size (in bytes) of decrypted data.

The Secure Asset is generated with `CM.4.secuasset.bat` in the `SBOOT` directory.

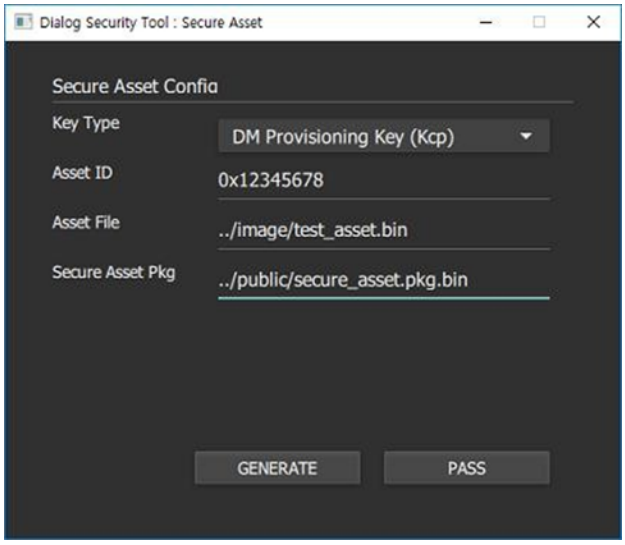

Figure 11. Encryption process of secure asset

The example code for decrypting a Secure Asset in Flash:

```
UINT32 status;
UINT32 assetsiz, encassetsiz;
UINT8  *asset;
UINT8  *dump_encasset_hex = NULL;
UINT32 address;

dump_encasset_hex = APP_MALLOC((512+48)); // header + asset

address = htoi(argv[1]);
encassetsiz = htoi(argv[2]);

status = sbrom_sflash_read(address, dump_encasset_hex, encassetsiz);

if (status == TRUE) {
    asset = CRYPTO_MALLOC(512);

    assetsiz = DA16X_Secure_Asset(2          // 1: Kpicv, 2: Kcp
                                , 0x00112233 // Asset ID
                                , (UINT32 *)dump_encasset_hex // secure asset
                                , encassetsiz // size of secure asset
                                , asset);     // decrypted asset

    if (assetsiz > 0) {
        CRYPTO_DBG_DUMP(0, asset, assetsiz);
    }
}

CRYPTO_FREE(asset);
```

3.7.2 Secure Storage

Secure Storage is a concept similar to the Secure Asset, but some features are different. Secure Storage is encrypted with a key derived from one of the following: user key, root key, Kcp, or Kpicv. It also supports full services to encrypt raw data and decrypt secure data, but the Secure Asset only supports a one-way function used to decrypt assets. The following example shows the functions and related definition items for Secure Asset.

```
typedef enum {

ASSET_USER_KEY = @, /**< User Key */

ASSET_ROOT_KEY = 1, 7**< HUK Key */

ASSET_KCP_KEY = 2, /**< DiiKey - Kep, asset(product) encryption key */
ASSET_KCE_KEY = 3, /**< DiKey - Kep, code encryption key */

ASSET_KPICV_KEY = 4, â€”/**< CMKey - Kpicy, asset(product) encryption key */
ASSET_KCEICV_KEY = 5, /**< CliKey - Kceicy, code encryption key */
} AssetKeyType_ts

Â® * \brief User Key Format{]
typedef struct {
uintÃ©_t â€œpkey; 7**< key data */
size_t keySize; /**< key size */
} AssetUserKeyData_t;

Â® * \brief Encrypted RunTime Asset Info[]

typedef struct {
uint32_t token; /**< ID for package provisioning */
uint32_t version; 7**< version info */

uint32_t assetSize;  /**< size of asset */
} AssetInfodata_t;

Â® * \brief ID code for RunTime Asset(]
#define CC_RUNASSET_PROV_TOKEN (@x416E7572UL
Â® * \brief Version code for RunTime Asset[]
#define CC_RUNASSET_PROV_VERSION @x10@@@UL

Â® * @brief Encrypt RuntimeAsset.[]
extern int32_t R_CC312_Secure_Asset_RuntimePack(AssetKeyType_t KeyType, uint32_t noncetype
Â» AssetUserkeyData_t *KeyData, uint32_t AssetID, char â€œtitle
Â» uint8_t *InAssetData, uint32_t AssetSize, uintÃ©_t *OutAssetPkgData);

Â® * @brief Decrypt RuntimeAsset.[]
extern int32_t R_CC312_Secure_Asset_RuntimeUnpack(AssetKeyType_t KeyType
Â» AssetUserkeyData_t *KeyData, uint32_t AssetID
> uint8_t *InAssetPkgData, uint32_t AssetPkgSize, uint8_t *OutAssetData);
```

Where:

- AssetKeyType_t
  Defines the type of the derived key stored in the OTP to be applied to the key derivation function CMAC to be used for encryption. ASSET_ROOT_KEY means Huk, ASSET_KCP_KEY means Kcp, and ASSET_KPICV_KEY means Kpicv. If the user-defined key is used in addition to the key stored in OTP, it should be defined as ASSET_USER_KEY and KeyData should be set as input value.
- AssetUserKeyData_t
  Specifies a user-defined key when ASSET_USER_KEY is used. The user-defined key defines 128/192/256 bits, pKey defines the buffer pointer of KeyData, and keySize defines 16/24/32 bytes, which means key length.

- R_CC312_Secure_Asset_RuntimePack()
  This function encrypts raw input data with an AES CCM method and has the following parameters:
  - KeyType

  Defines the type of decryption key to use for encryption.
  - Noncetype

  Defines how to generate the nonce information used in the encryption process. '0' is the Nonce generated by TRNG, and '0xFFFFFFFF' is the Nonce generated by PRNG.
  - KeyData

  Specifies the parameter to input User Key when KeyType is defined as ASSET_USER_KEY.
  - AssetID

  Specifies the ID information used in the encryption process.
  - Title

  Specifies the title information of the Runtime Asset Package.
  - InAssetData

  Specifies the data pointer of the raw data to be encrypted.
  - AssetSize

  Specifies the size of the raw data and must be defined as 16 bytes multiple for AES, which is a block cipher.
  - OutAssetPkgData

  Specifies the data pointer of the encrypted Runtime Asset Package. Since the function does not perform internal memory allocation, the data buffer for the output data should be pre-allocated and allocated to Raw Data Size + 48 bytes, considering 48 bytes of information field to be additionally tagged.
  If the Return Value is less than 0, it means error. If the Return Value is larger than 0, it means size information of output data OutAssetPkgData.
- R_CC312_Secure_Asset_RuntimeUnpack()
  This function decrypts the encrypted input Runtime Asset Package, and the input parameter needs to input the encryption parameter applied to function FC9K_Secure_Asset_RuntimePack().
  - KeyType

  This should match the type of decryption key used in encryption.
  - KeyData

  If KeyType is defined as ASSET_USER_KEY, it should match key information used as User Key.
  - AssetID

  This should match the ID information used for encryption.
  - InAssetPkgData

  The data pointer of the Runtime Asset Package to be decoded.
  - AssetPkgSize

  The size of the Runtime Asset Package, which means Raw Data Size + 48 bytes.
  - OutAssetData

  The data pointer of the decoded raw data, and the size is the raw data size.
  If the Return Value is less than 0, it means an error. If the Return Value is larger than 0, it means size information of output data OutAssetData.

See example code to implement Secure Storage in Flash that uses the Runtime Pack/Unpack function.

Example 1: Secure asset runtime APIs

```
> int32_t R_CC312_Secure_Asset_RuntimePack(AssetkeyType_t KeyType, Uint32_t noncetype
Â» AssetUserkeyData_t *KeyData, Uint32_t AssetID, char *title
Â» uint8_t *InAssetData, uint32_t AssetSize, uints_t *OutAssetPkgData)

Gint32E re = CC_OK;

uint32t i;

CCUtilAesCmacResult_t keyProv = { @ }5

uint&_t  dataIn[CC_UTIL_MAX_KDF_SIZE_IN_BYTES] = { @ }5
wint32_â,¬  dataInSize = CC_UTIL_MAX_KDF_SIZE_IN_BYTES;
uint&_t â€”_provlabel = 'P
CCRunAssetProvPkg_t * pAssetPackage = NULL;
mbedtls_ccm_context â€” ccmCtx;

uint@_t * pencRunAsset;

/* Validate Inputs */

) if ((InAssetData == NULL) ||
(OutAssetPkgData NULL) ||
/*(AssetSize > CC_ASSET_PROV_MAX_ASSET_SIZE) ||*/
(AssetSize == @) ||
((AssetSize % CC_ASSET_PROV_BLOCK_SIZE) != @) ||
(((UtilkeyType_t) KeyType) > UTIL_KCEICV_KEY) )

CC_PAL_LOG_ERR("Invalid params");
return (int32_t) CC_UTIL_ILLEGAL_PARAMS_ERROR;

}

re = CCProd_Init();
> if (re != CC_OK) {
CC_PAL_LOG_ERR("Failed to CCProd_Init Ox%x\n", rc);
goto cc312_sa_pack_end_raw;

}

pAssetPackage = (CCRunAssetProvPkg t *) OutAssetPkgData;
pencRunAsset = (uint8_t *) (OutAssetPkgData+ sizeof (CCRunAssetProvPkg_t));
```

Example 2: Secure asset runtime APIs

```
/* fill asset size, must be multiply of 16 bytes */
pAssetPackage->assetSize = AssetSize;

/* Â£411 package token and version */
pAssetPackage->token = CC_RUNASSET_PROV_TOKEN;
pAssetPackage->version = CC_RUNASSET_PROV_VERSION;

if( title != NULL ){
CRYPTO_MEMCPY(pAssetPackage->reserved, title
Â» (CC_ASSET_PROV_RESERVED_WORD_SIZE*sizeof(Uint32_t)) );
}

/* Generate dataIn buffer for CMAC: iteration || 'P' || @x@@ || asset Id || @x8e
since deruyed key is 128 bits we have only 1 iteration */
re = UtilCmacBuildDataForDerivation(&provLabel, sizeof (provlabel),
(uint8_t *) SAssetID, sizeof(AssetID),
dataIn, (size_t *) &dataInSize,
(size_t)CC_UTIL_AES_CMAC_RESULT_SIZE_IN_BYTES);
if (re != @) {
CC_PAL_LOG_ERR("Failed UtilCmacBuildDataForDerivation Ox%x", rc);
CRYPTO_MEMSET(pAssetPackage, @, sizeof(CCRunAssetProvPkg_t))3
goto cc312_sa_pack_end_raw;

dataIn[@] = 1; // only 1 iteration
re = UtilCmacDerivekey(((UtilkeyType_t) KeyType),
((CCAesUserkeyData_t *) KeyData),
dataIn, dataInSize,
keyProv);
if (re != 0) {
CC_PAL_LOG_ERR("Failed UtilCmacDeriveKey @x%x", rc);
CRYPTO_MEMSET(pAssetPackage, @, sizeof (CCRunAssetProvPkg_t));
goto cc312_sa_pack_end_raw;

/* Decrypt and authenticate the BLOB */
mbedt1s_ccm_init(&ccmCtx);
```

Example 3: Secure asset runtime APIs

```
re = mbedtls_ccm_setkey(&ccmCtx, MBEOTLS CIPHER ID AES, keyProv, CC_UTIL_AES_CMAC_RESULT_SIZE_IN
BYTES * CC_BITS_IN BYTE);
if (re t= 0) {
CC_PAL_LOG_ERR("Failed to mbedtis ccm setkey @x%x\n", re);
CRYPTO_MEMSET(pAssetPackage, @, sizeof(CCRunAssetProvPkg_t))5
goto cc312_sa_pack_end;

+

i f( noncetype == OxFFFFFFFF ){
for(i = 0; i < CC_ASSET_PROV_NONCE_SIZE; i++ ){
pAssetPackage->nonce[i] = (uint8_t) embcrypto_random();

Jelset
wints_t â€œpkey:
wints_t "pr

Uint32_â,¬ â€œpentrsres
wint32_t sourcesize;
int32_t â€œpRndviorkBuf fs

pRndorkBuff = (UERES2IE â€œ) CRYPTO_MALLOC(CHPU_NORKSPACE_MINIMUM_SIZE)5
pkey = (uint_t *) CRYPTO MALLOC(CC_PROD_AES Key256Bits_SIZE_IN BYTES);
plv = (uint_t *) CRYPTO_MALLOC(CC_PROD_AES_IV_COUNTER_SIZE_IN_BYTES);

F( (pRndhiorkBuff == MULL) [| (pKey == NULL) || (pT
iF( pRndiorkBuff != NULL ){
â€˜CRYPTO_FREE(pRndiorkBuff) ;

NULL) ){

+

G#( pkey != NULL ){
â€˜CRYPTO_FREE(pKey) 5

}

Gf( plv t= MULL ){
â€˜CRYPTO_FREE(pIV)3

Fe = CC_UTIL_FATAL_ERROR;
goto cc312_sa_pack end;
+

pRndbiorkBuff[@] = (UERESZIE)exFc9cc312; // hidden tog
```

Example 4: Secure asset runtime unpack API

```
| * R_CC312_Secure_Asset_RuntimeUnpack( ){]

'int32_t R_CC312_Secure_Asset_RuntimeUnpack(AssetKeyType_t KeyType
Â» AssetUserkeyData_t * KeyDafa, uint32_t AssetID
Â» uint8_t * InAssetPkgData, uint32_t AssetPkgSize, uintÃ©_t * OutAssetData)

{
CCUtilAesCmacResult_t keyProv {0}5
CCRunAssetProvPkg t * pAssetPackage = NULL;
mbedt1s_ccm_context cemCtx3
uint32_t re = CC_OK;
uint32_t dataInSize = CC_UTIL_MAX_KDF_SIZE_IN_BYTES;
uintÃ©_t dataIn[CC_UTIL_MAX_KDF_SIZE_IN_BYTES] = {0};
uintÃ©_t provlabel
uintÃ©_t * pencRunAsset;
bool sbstyle = 0;

/* Validate Inputs */
if ((InAssetPkgData == NULL) ||
(QutAssetData == NULL) ||
(((UtilkeyType_t)KeyType) > UTIL_KCEICV_KEY) )

CC_PAL_LOG_ERR("Invalid paramsâ€ );
return (int32_t)CC_UTIL_ILLEGAL_PARAMS_ERROR;

}

re = CCProd_Init();

if (re != CC_OK) {
CC_PAL_LOG_ERR("Failed to CCProd_Init Ox%x\n", rc);
return (int32_t)res

+
pAssetPackage = (CCRunAssetProvPkg t *) InAssetPkgData;
pencRunAsset = (uint&_t *) (InAssetPkgData + sizeof(CCRunAssetProvPkg _t));

/* Validate asset size, must be multiply of 16 bytes */
if ((pAssetPackage->assetSize @) II
```

3.7.3 Secure Storage on Flash

RA6W1 and RA6W2 use RM_VEE_FLASH and RM_PSA_CRYPTO, and the following are the APIs used for encrypted data into the RM_VEE_FLASH. Secure assets are saved to Flash using RM_VEE_FLASH.

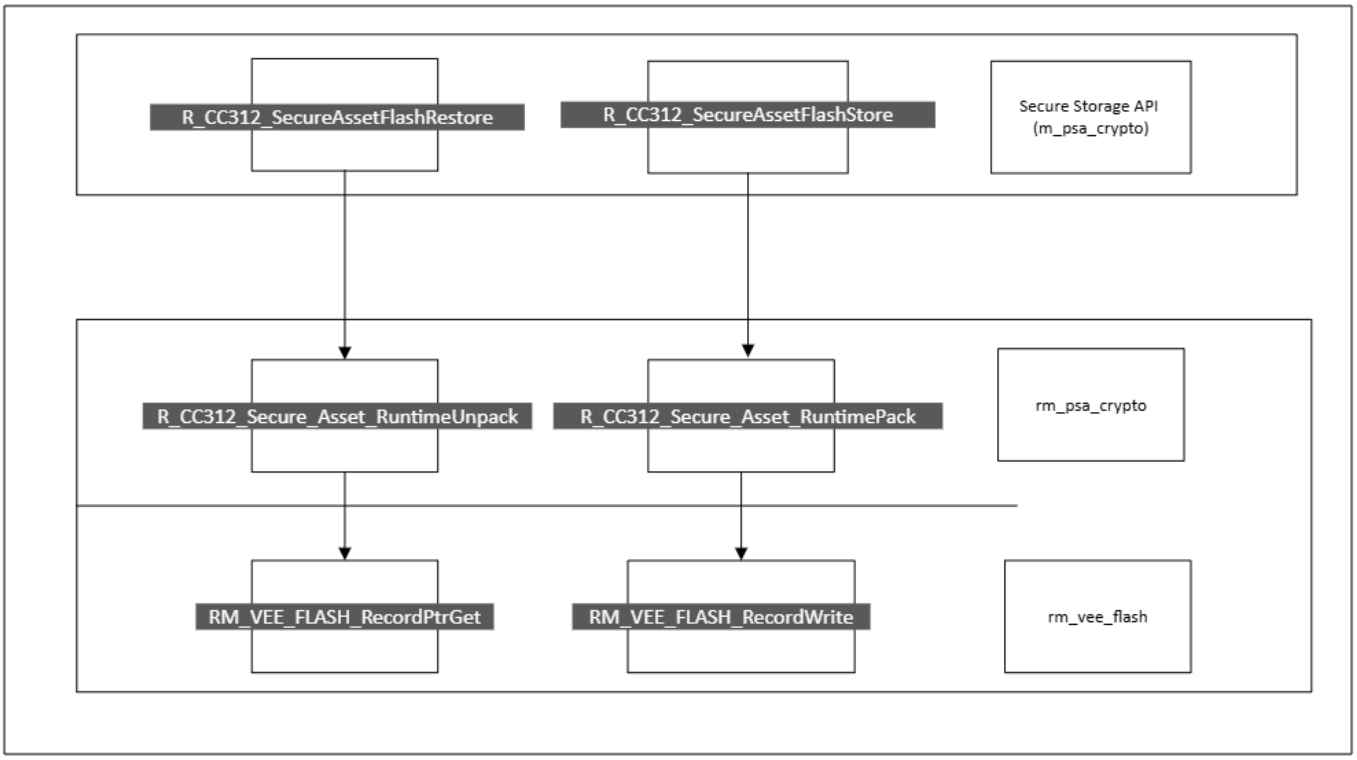

Figure 12. Secure APIs used in RM_VEE_FLASH

R_CC312_SecureAssetFlashStore and R_CC312_SecureAssetFlashRestore encrypt assets in RM_VEE. These functions use the function R_CC312_Secure_Asset_RuntimePack for encrypting the asset and storing the data in rm_vee_flash using the function RM_VEE_FLASH_W_RecordWrite.
The R_CC312_Secure_Asset_RuntimePack function uses binary data encryption. The following is a description of the KeyType argument for this function:

```
typedef enum  {
    ASSET_USER_KEY = 0,      /**< User Key */
    ASSET_ROOT_KEY = 1,      /**< HUK Key */
    ASSET_KCP_KEY = 2,       /**< DMKey - Kcp, asset(product) encryption key */
    ASSET_KCE_KEY = 3,       /**< DMKey - Kcp, code encryption key  */
    ASSET_KPICV_KEY = 4,     /**< CMKey - Kpicv, asset(product) encryption key */
    ASSET_KCEICV_KEY = 5,    /**< CMKey - Kceicv, code encryption key  */
} AssetKeyType_t;
```

ASSET_KCP_KEY is used as the key type of R_CC312_Secure_Asset_RuntimePack. When the device completes, securing provisioning key is safely stored in ASSET_KCP_KEY. Therefore, the encrypted data is stored through the key Using VEE.

Figure 13 explains the data encryption process.

- Kprov = AES-CMAC(Kcp, 'P' || 0x00 || asset id || 0x80)
- Nonce = os.urandom(12) // TRNG
- Wrapped key = AES.CCM (Kprov, Nonce, Tag, Data)

Secure Storage is encrypted with a key derived from one of the followings: user key, root key, ASSET_KCP_KEY. It also supports full services to encrypt raw data and decrypt secure data, but the Secure Asset only supports one-way function used to decrypt assets.

- R_CC312_Secure_Asset_RuntimePack
- R_CC312_Secure_Asset_RuntimeUnpack.

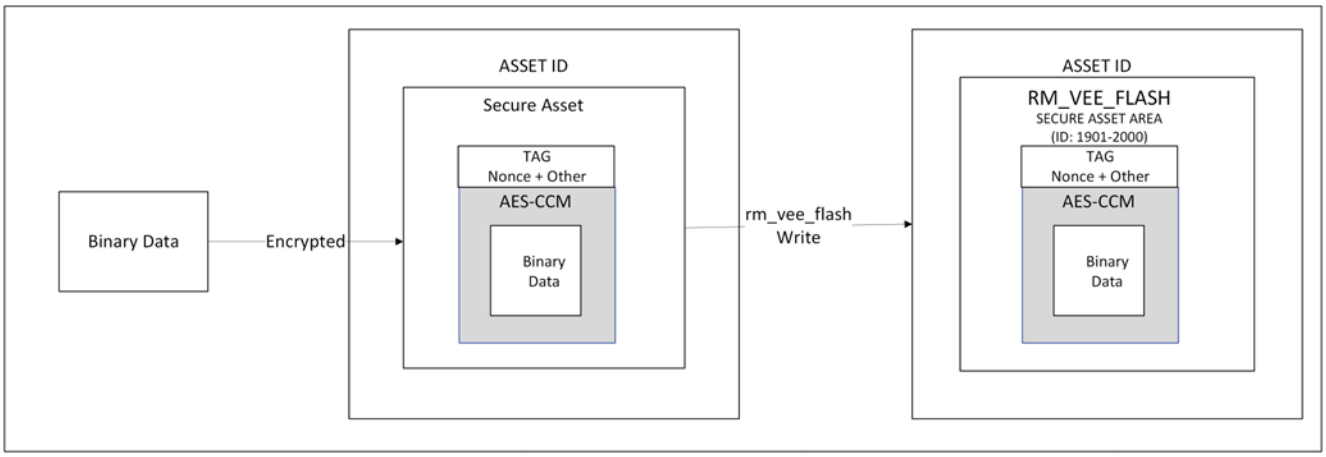

Figure 13. Stored encryption data

3.7.3.1 CM LCS

MbedTLS APIs are used for cryptographic functions in the RA6W1/RA6W2. MbedTLS is an open-source SSL library that enables you to include cryptographic and SSL/TLS capabilities in their embedded products, with a minimal coding footprint.

You can choose between hardware-accelerated cryptographic operations and the software cryptographic implementation of Mbed TLS for each feature supported by both Mbed TLS and CryptoCell-312:

- The Mbed TLS cryptographic implementation provides an interface to the standard cryptographic operations. For example, AES, RSA, or ECC.
- The dedicated CryptoCell-312 APIs provides an interface to the non-standard or specific CryptoCell-312 operations. For example, key derivation using HUK.

Mbed TLS and CryptoCell-312 are flexible in terms of which features are compiled in each. To control which components are Mbed TLS-based or CryptoCell-312-based, you must edit the `config-cc312.h` configuration file. This file is located in `crypto/inc/mbedtls/config.h`. It includes all the flags that are supported by Mbed TLS, with the additional XXX_ALT definitions. These XXX_ALT definitions are for the components that are accelerated by the hardware.

By default, Renesas Electronics SDK comes with the minimal required features that CryptoCell-312 accelerates. See the open-source Mbed TLS documentation for how to use Mbed TLS APIs.

Table 9 shows the supported hardware acceleration crypto algorithms in the RA6W1/RA6W2.

Table 9. Hardware acceleration crypto algorithms

Algorithm	Mode	Key sizes
AES	ECB, CBC, CTR, OFB, CMAC, CBC-MAC, AES-CCM, AES-CCM*, AES-GCM	128 bits, 192 bits, and 256 bits
AES key wrapping	N/A	All
Chacha and Chacha-Poly1305	N/A	N/A
Diffie-hellman ANSI X9.42-2003: Public Key Cryptography for the Financial Services Industry: Agreement of Symmetric Keys Using Discrete Logarithm Cryptography Public-Key Cryptography Standards (PKCS) #3: DiffieHellman Key Agreement Standard.	N/A	1024 bits, 2048 bits, and 3072 bits
ECC key generation	N/A	NIST curves and 25519 curves
ECIES	N/A	NIST curves and 25519 curves
ECDSA	N/A	NIST curves and ED25519
ECDH	N/A	NIST curves and 25519 curves
Hash	SHA1, SHA224 and SHA256	N/A
HKDF	N/A	N/A
HMAC	SHA1, SHA224 and SHA256	N/A
KDF NIST SP 800-108: Recommendation for Key Derivation Using Pseudorandom Functions	CMAC or HMAC	N/A
RSA PKCS#1 operations Public-Key Cryptography Standards (PKCS) #1 v2.1: RSA Cryptography Specifications. Public-Key Cryptography Standards (PKCS) #1 v1.5: RSA Encryption.	Encryption and signature schemes	2048 bits, 3072 bits, and 4096 bits
RSA key generation	N/A	2048 bits, 3072 bits

4. Security Tool

The Security tool (the SBOOT tool) facilitates the generation of keys and assets for the RA6W1 and RA6W2 devices. The tool is designed to produce secret keys, certificates, and secure binary images specifically for these platforms. Its four primary functions are as follows:

- Generation of RoT (Hbk0, Hbk1) and CM/DM secret keys (Kpicv, Kceicv, Kcp, and Kce), including CMPU and DMPU binaries which incorporate all keys intended for programming into OTP memory.
- Construction of Secure Boot images, such as secure bootloader and RTOS images, compatible with the RA6W1/RA6W2.
- Creation of Secure Debug certificates and corresponding images.
- Production of RMA certificates and associated images.

To open the SBOOT tool:

1. Unzip the `SBOOT` directory, navigate to the `SBOOT/image`, and then open the `non_secure_cfg.xml.AT25SL641-8MB` file in an editor.
2. Edit line 28, so that the version name complies with the following convention:
   Five segments, separated by four hyphens, for example:
   `<version>RA6W1-RRQ61001-8-0-3</version>`
   Before editing:

```
<image binfile="indium.bin" offset="0x00002000">
    <version>1.0.0</version>
    <timestamp>0x5939110D</timestamp>
```

Figure 14. Before editing non_secure_cfg xml

   After editing:

```
<image binfile="indium.bin" offset="0x00002000">
    <version>RA6W1-RRQ61001-8-0-3</version>
    <timestamp>0x5939110D</timestamp>
```

Figure 15. After editing non_secure_cfg xml

3. Navigate back to the `/SBOOT` directory, and then open `CM.1.secuman.bat`.

Figure 17 shows the main window of the SBOOT tool when running `CM.1.secuman.bat` from the `/SBOOT` directory (Figure 16).

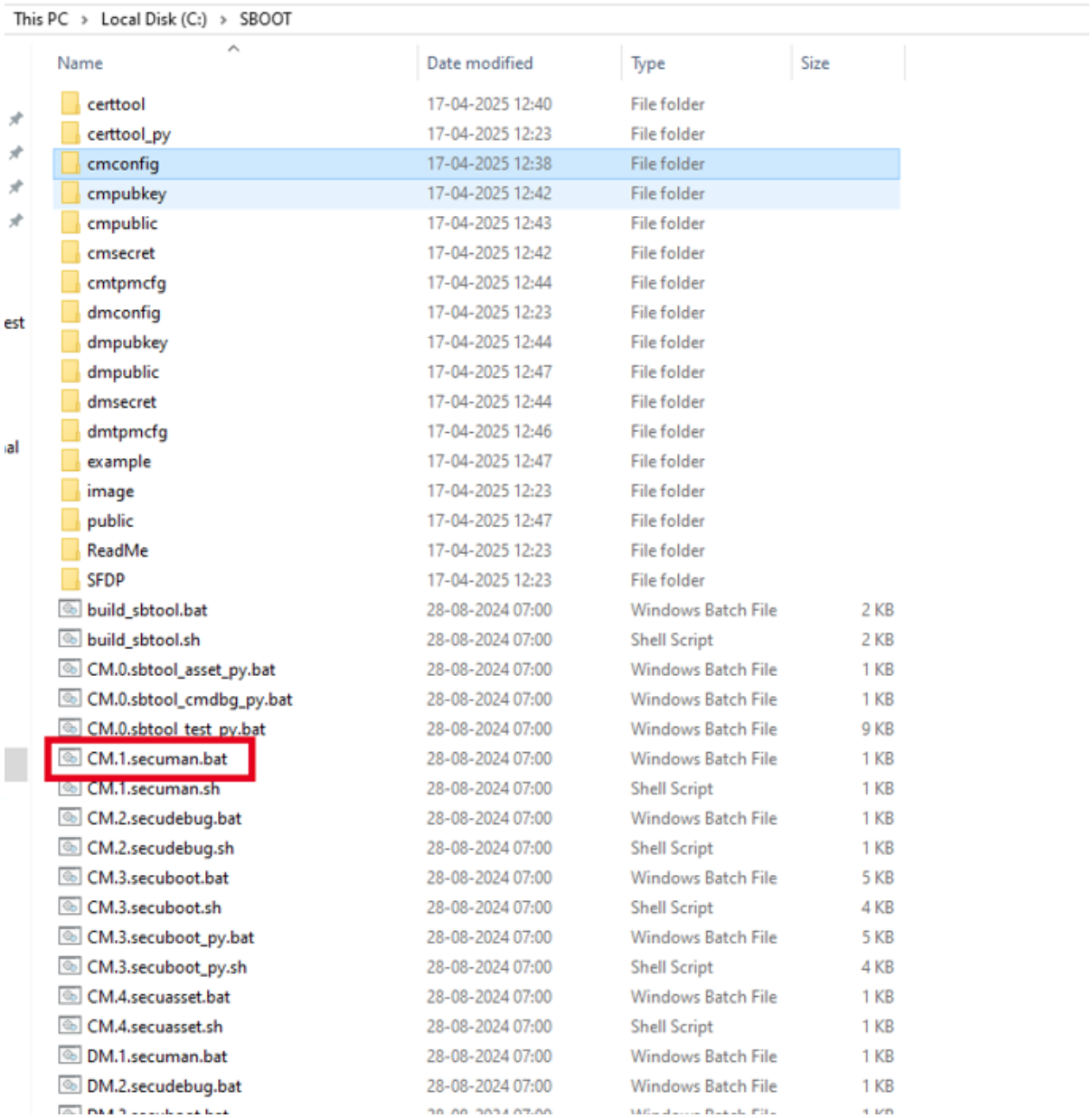

Figure 16. SBOOT tool directories

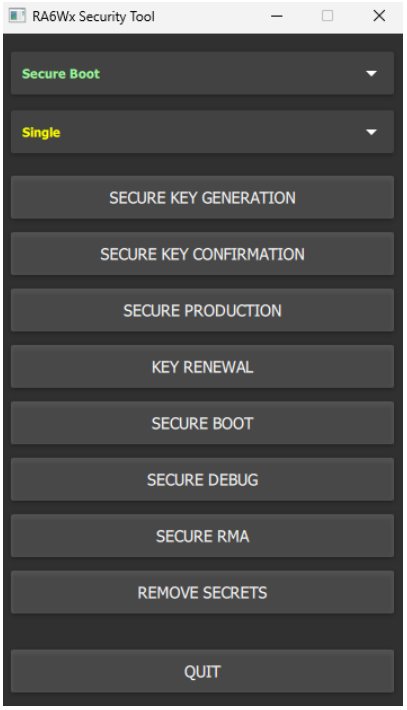

Figure 17. Main window of the SBOOT tool

4.1 Role Selection

In the main window, select the SBOOT tool operating role from the drop-down menu. Three options are available:

- **Single Manager**
  The "Single" Manager holds primary responsibility for generating and administering all secret keys associated with the product. Exclusive authority to generate, renew, or revoke these secret keys resides with the Single Manager. Notably, the private key corresponding to the Root of Trust (Hbk0 and Hbk1) stored in the OTP memory must be securely managed and maintained by the Single Manager.
  The Single Manager is entrusted with the responsibility of safeguarding the private key and must not disclose it under any circumstances. Any exposure of the private key compromises security, necessitating the recall of products containing the corresponding RoT in the OTP. Therefore, exercise caution when assigning the "Single" manager role to a developer.
- **SB Publisher**
  The "SB Publisher" role is responsible for generating the third certificate—such as the content certificate—which is required for Secure Boot within a three-level certificate scheme, and for rebuilding secure bootable images (RTOS and other images) with this certificate. Only the Secure Boot menu is accessible under this role. The primary duty of this position is to eliminate the debug certificate from the image following Secure Debug operations. After Secure Debug, images contain a debug certificate, which activates debug interfaces. This role should be used to remove the debug certificate and produce exclusively Secure Boot images that restrict debug interfaces, thereby enhancing security.
- **SB/SD Publisher**
  The "SB/SD Publisher" role is responsible for creating the third certificate—such as the content certificate—which is necessary for Secure Boot in a three-level certificate system and for rebuilding Secure Boot images (including RTOS and other images). Additionally, the "SB/SD Publisher" generates the Debug certificate required for Secure Debug, using the SoC-ID of the target device to enable its debug interface (the JTAG port).

4.2 Secure Key Generation

To initiate secure key creation, first generate the `icv_request_pck.bin` and the `oem_request_pkg.bin` files using the **Secure Key Generation** button in the SBOOT tool. These files are then sent to a secure Renesas server to trigger the creation of response packages, which will be used later.
To generate request files:

1. Create an empty dummy file saved as `RA6Wx_cache.bin` and save it in the `/SBOOT/image` directory.
2. Create an empty dummy file saved as `krtl.key` and save it in the `/SBOOT/cmsecret` directory.
3. In the SBOOT tool, click **Secure Key Generation**.

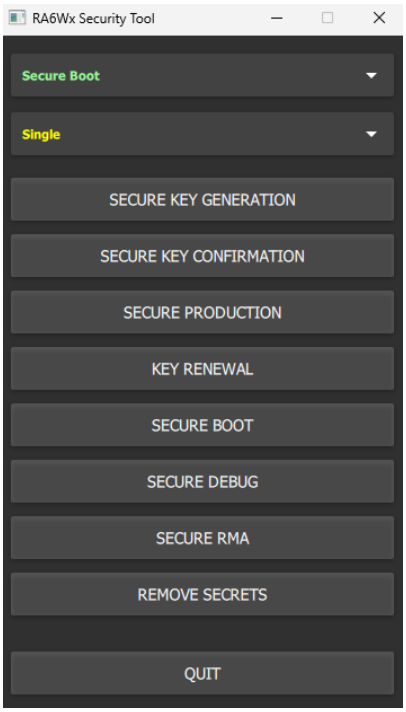

Figure 18. Secure Key Generation button

4. When prompted, click **Yes to All**.
5. Verify that the `icv_request_pkg.bin` and the `oem_request_pkg.bin` files were generated in the `/SBOOT/public` directory.

4.3 Security Confirmation

During the Security Confirmation phase, response packages are generated.

NOTE
The current revision of the SBOOT tool has the Secure Key Confirmation button available in the main menu. Skip this option and use the procedure described in this section.

To generate response packages:

NOTE
You must request user account creation by contacting customer support. Customer support then creates the account and provides the initial login credentials.

1. Log into the official website: https://sekeool.com/.
2. Upload the `icv_request_pkg.bin` and `oem_request_pkg.bin` files in the order specified by the online tool, as shown in Figure 19.
3. After the upload is complete, generate and download both the `dm_icv_response_pkg.bin` and the `cm_icv_response_pkg.bin` files.

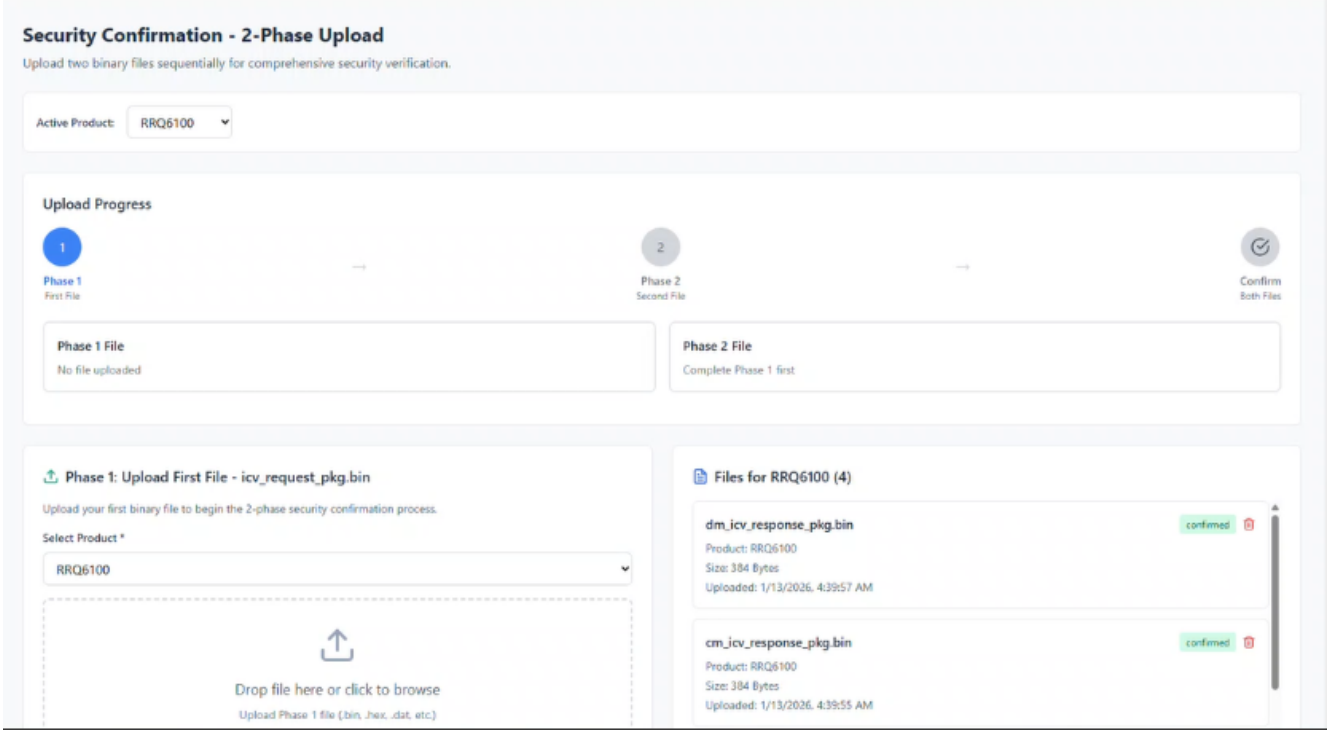

Figure 19. Upload icv_request_pkg.bin

4. Place the `dm_icv_response_pkg.bin` and `cm_icv_response_pkg.bin` files in the `SBOOT/public` directory.

4.4 Secure Production

Secure Production is responsible for generating all secret keys, including CM keys, DM keys, and those required for the second and third certificates, for example, the multi-level certificate chain used by Secure Debug and related security services. Corresponding certificate chains are then created using these keys to facilitate the generation of Secure Boot and Secure Debug images.

NOTE
It is essential to build the SDK prior to initiating the Secure Production process.

Table 10 lists the files generated in the Secure Production stage.

Table 10. Secret keys for secure production

Items	CM/DM keys	Directory	Generated files
CM keys	CM keys	cmsecret	OTP keys: cmkey_pair.pem, kceicv.bin, kpicv.bin
			Private keys for Secure Boot: cmissuer_keypair.pem, cmpublisher_keypair.pem
			Private keys for Secure Debug: cmenabler_keypair.pem, cmdeveloper_keypair.pem
DM keys	DM keys	dmsecret	OTP keys: dmkey_pair.pem, kce.bin, Kcp.bin
			Private keys for Secure Boot: dmissuer_keypair.pem, dmpublisher_keypair.pem
			Private keys for Secure Debug: dmenabler_keypair.pem, dmdeveloper_keypair.pem
Certificates for Secure Boot	with CM keys	cmpublic	sboot_hbk0_3lvl_key_chain_issuer.bin, sboot_hbk0_3lvl_key_chain_publisher.bin
	with DM keys	mpublic	sboot_hbk1_3lvl_key_chain_issuer.bin, sboot_hbk1_3lvl_key_chain_publisher.bin,

Items	CM/DM keys	Directory	Generated files
			Content certificates for UEboot and RTOS images
Certificates for Secure Debug	with CM keys	cmpublic	sdebug_hbk0_3lvl_key_chain_enabler.bin, sdebug_hbk0_3lvl_key_chain_developer.bin
	with DM keys	dmpublic	sdebug_hbk1_3lvl_key_chain_enabler.bin, sdebug_hbk1_developer_pkg.bin

To enter CM and DM secret keys into OTP memory, special binaries called the CMPU package and DMPU package are also generated after Secure Production, containing the items shown in Table 10.

4.4.1 Initializing Secure Production

NOTE

Before you start, verify that:

- The `RA6Wx_cache.bin` and the `krtl.key` dummy files are created and saved in the correct directories, see Section 4.2 Secure Key Generation. Without the correct platform key in that location, Secure Production cannot proceed.
- The `cm_icv_response_pkg.bin` and the `dm_icv_response_pkg.bin` files are obtained and placed in the `SBOOT/public` directory. See Section 4.3 Security Confirmation.

To initialize Secure Production, proceed according to the instructions in the following sections.

4.4.2 Generate CMPU and DMPU Hex Values

1. In the SBOOT tool, click **Secure Production.**
2. When prompted, click **Yes to All**.

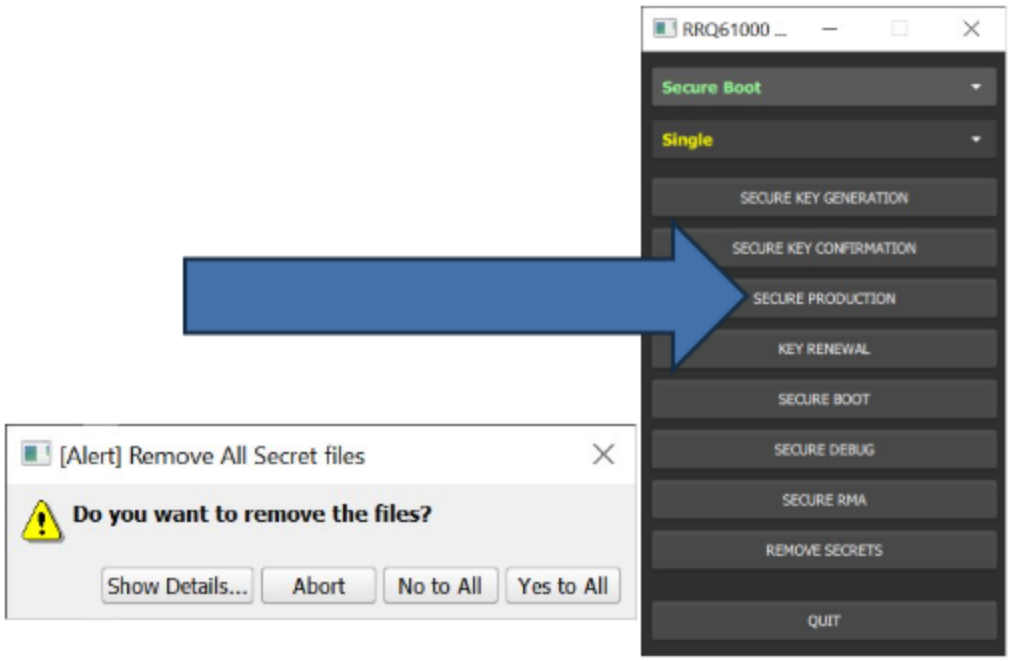

Figure 20. Secure Production

3. In the **Secure Parameters** window (Figure 21), adjust the NV count value relative to CM, if necessary. Click **Update** to continue.

NOTE
NV count is an anti-rollback counter. It prevents loading of older CMPU or DMPU security packages after secure provisioning.

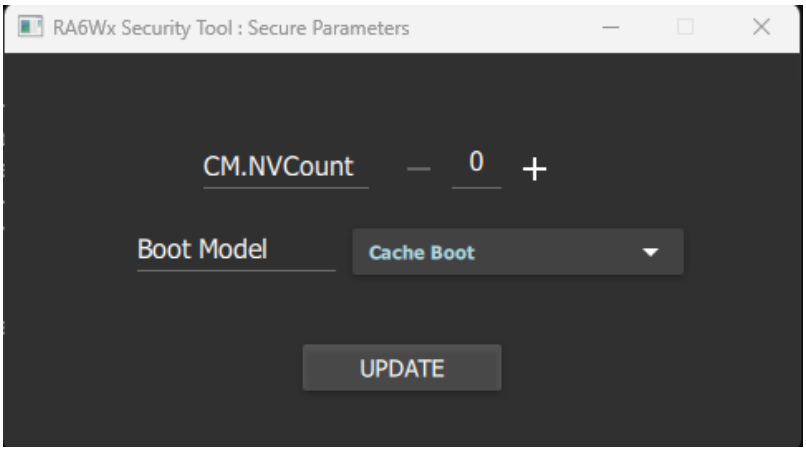

Figure 21. Secure Production – CM NV count selection

4. In the **Enabler** window (Figure 22), click **Update**.

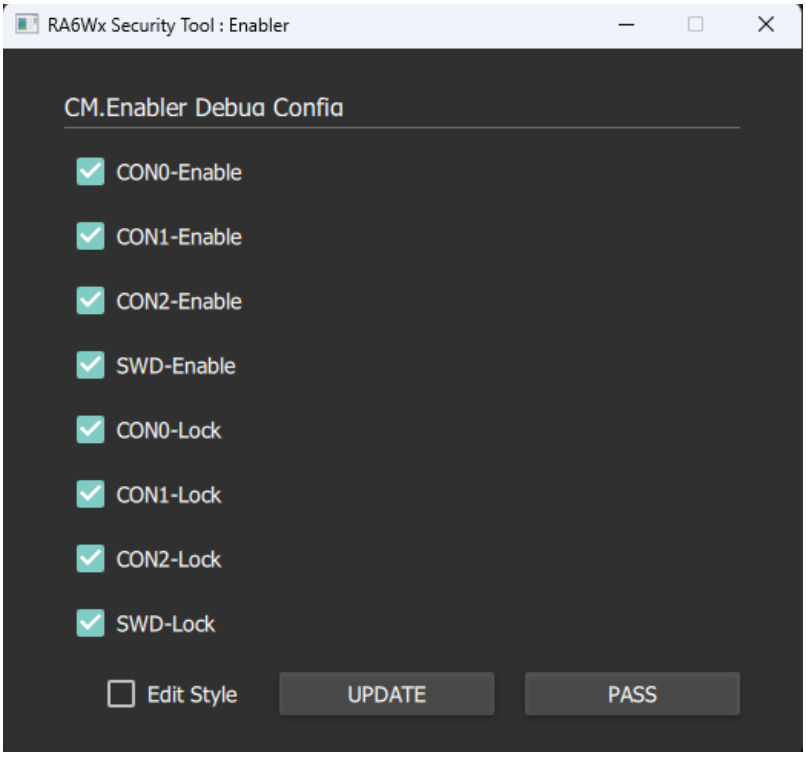

Figure 22. Secure Production – debug config selection

5. Continue the secure DM configuration process (Figure 23):
   a. In the **Secure Parameters** window, adjust the NV count value relative to DM, if necessary, and then click **Update**.
   b. In the **Enabler** window, click **Update**.

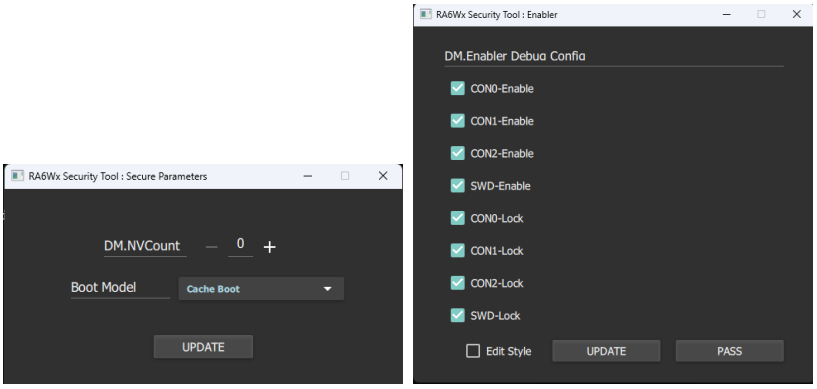

Figure 23. Secure Production – DM configuration process

6. In the **RTOS.cache** window (Figure 24), from the **Encrypt Scheme** drop-down menu, select **NO_IMAGE_ENC** when booting without security. CMPU and DMPU processes must first run in a non-secure environment.

NOTE
For the RA6W2 platform, rename the COMP1 and IMAGE name from RA6W1 to RA6W2.

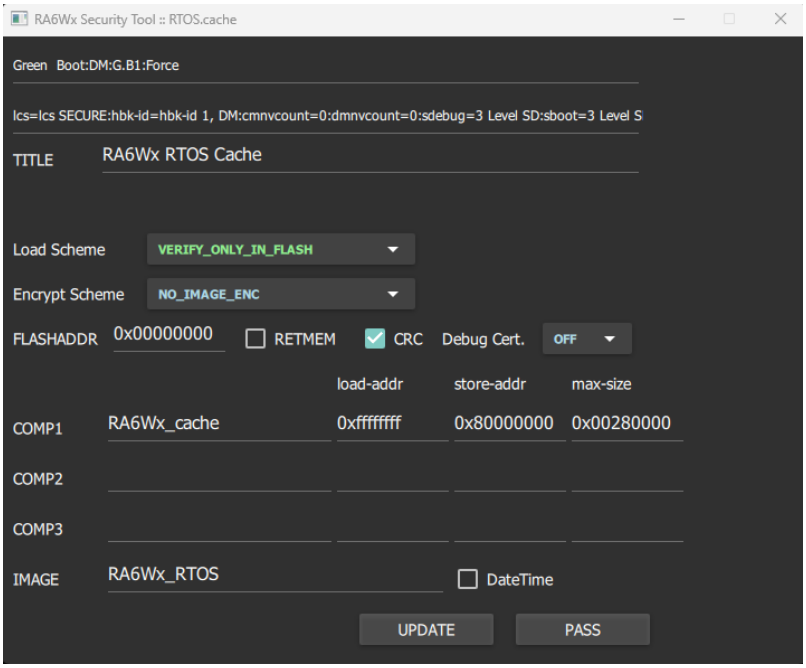

Figure 24. Secure Production – RTOS cache config

7. Click **Pass**. New binaries and txt files are generated in the `/SBOOT/public` directory, see Figure 25.

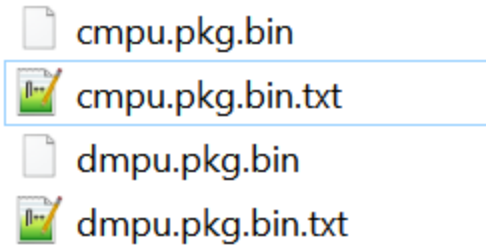

Figure 25. cmpu and dmpu hex files

4.4.3 Update the CMPU and DMPU Hex Value

1. Copy the hex values present in the `cmpu.pkg.bin.txt` and `dmpu.pkg.bin.txt` into the `rm_cli_sbrom.c` e² studio file.
   File location: `\e2_studio\workspace\<Project_Name >\rrq\fsp\src\rm_cli\rm_cli_sbrom.c`
2. Make the following changes in the file:
   a. Enable the `SUPPORT_SECURE_PRODUCTION` flag.

```
/* NOTICE !
 * We will release SDK with 'SUPPORT_SECURE_PRODUCTION = 0' to protect Secure OTP.
 * After keeping in mind that once this process has begun and then it cannot be cancelled,
 * please change 'SUPPORT_SECURE_PRODUCTION = 1' to continue.
 */

#define SUPPORT_SECURE_PRODUCTION    (1)
```

   b. Update the hexadecimal values shown in Figure 26 and Figure 27 according to the new **cmpu hex** and the **dmpu hex** values.

```
//-------------------------------------------------------------
#if (SUPPORT_SECURE_PRODUCTION == 1)
__ALIGNED(4) static const unsigned char cmpu_hex_list[] = {
0x01, 0x00, 0x00, 0x00,         //uniqueDataType
0x4d, 0x12, 0xf9, 0x04, 0x6c, 0xda, 0xb3, 0xfe, 0x4f, 0x9f, 0xae, 0xb1, 0x26, 0xf2, 0x6b, 0x5a,
       //uniqueBuff
0x02, 0x00, 0x00, 0x00,         //kpicvDataType
0x64, 0x6f, 0x72, 0x50, 0x00, 0x00, 0x01, 0x00, 0x10, 0x00, 0x00, 0x00, 0x31, 0x76, 0x65, 0x52,
0x32, 0x76, 0x65, 0x52, 0xe8, 0x4b, 0x07, 0x0a, 0x21, 0x48, 0xbe, 0x1a, 0xc1, 0xbc, 0xd9, 0x18,
0xee, 0x1b, 0x5a, 0xa4, 0xa8, 0x58, 0x3a, 0x92, 0x06, 0x13, 0x1d, 0xa7, 0x6a, 0x1f, 0x89, 0xd8,
0xb9, 0x50, 0x2f, 0x29, 0x9a, 0xa4, 0xb3, 0x22, 0x38, 0x3f, 0x9d, 0x8a, 0x2f, 0x39, 0xe9, 0x97,
       //kpicv
0x02, 0x00, 0x00, 0x00,         //kceicvDataType
0x64, 0x6f, 0x72, 0x50, 0x00, 0x00, 0x01, 0x00, 0x10, 0x00, 0x00, 0x00, 0x31, 0x76, 0x65, 0x52,
0x32, 0x76, 0x65, 0x52, 0x4a, 0xde, 0x31, 0x78, 0xe7, 0x35, 0x9e, 0xb9, 0xb3, 0x26, 0xd2, 0x5e,
0xeb, 0x91, 0x93, 0x12, 0x52, 0x4f, 0x35, 0xa7, 0x23, 0x8d, 0x5d, 0x1b, 0x06, 0x80, 0xd5, 0x12,
0x84, 0x53, 0x25, 0x03, 0x70, 0x44, 0x25, 0x05, 0xa6, 0x85, 0x9c, 0xe4, 0xdf, 0x96, 0x8d, 0x42,
       //kceicv
0x00, 0x00, 0x00, 0x00,         //icvMinVersion
0x00, 0x00, 0x00, 0x00,         //icvConfigWord
0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0x00, 0x00, 0x00, 0x00,
       //icvDcuDefaultLock
};
```

Figure 26. Changes with the new hex values – cmpu hex

```
__ALIGNED(4) static const unsigned char dmpu_hex_list[] = {
0x01, 0x00, 0x00, 0x00,         //uniqueDataType
0xa7, 0x49, 0x63, 0x2c, 0xf6, 0xed, 0x27, 0x27, 0x40, 0x12, 0xd8, 0x8e, 0x33, 0x35, 0x66, 0xba,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
       //uniqueBuff
0x02, 0x00, 0x00, 0x00,         //kcpDataType
0x64, 0x6f, 0x72, 0x50, 0x00, 0x00, 0x01, 0x00, 0x10, 0x00, 0x00, 0x00, 0x31, 0x76, 0x65, 0x52,
0x32, 0x76, 0x65, 0x52, 0xa5, 0xfc, 0x9a, 0xb4, 0x01, 0x0b, 0x7b, 0xdb, 0x20, 0x4a, 0xbe, 0x95,
0xc1, 0x9c, 0x95, 0x3c, 0xcf, 0x4d, 0xc7, 0x75, 0x3f, 0x4f, 0x1d, 0x58, 0x2c, 0xc7, 0xbc, 0x18,
0x3c, 0x8a, 0x62, 0x95, 0x96, 0x0d, 0xa5, 0x28, 0x56, 0x2c, 0x9e, 0x45, 0xee, 0x58, 0xe6, 0xec,
       //kcp
0x02, 0x00, 0x00, 0x00,         //kceDataType
0x64, 0x6f, 0x72, 0x50, 0x00, 0x00, 0x01, 0x00, 0x10, 0x00, 0x00, 0x00, 0x31, 0x76, 0x65, 0x52,
0x32, 0x76, 0x65, 0x52, 0xfb, 0xb6, 0xde, 0xd1, 0xfe, 0x6f, 0x78, 0x06, 0xff, 0x32, 0x4f, 0x18,
0x6d, 0xb4, 0x11, 0x16, 0xaf, 0xe5, 0x40, 0xbc, 0x38, 0x56, 0x47, 0x7f, 0xb9, 0xb6, 0x74, 0xbc,
0x94, 0xe7, 0xf6, 0x44, 0x24, 0x63, 0x14, 0xdb, 0x85, 0xff, 0xc7, 0x5b, 0x2f, 0x43, 0xdf, 0x23,
       //kce
0x00, 0x00, 0x00, 0x00,         //oemMinVersion
0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0x00, 0x00, 0xff, 0xff,
       //oemDcuDefaultLock
};
#endif
```

Figure 27. Changes with the new hex values – dmpu hex

3. Initiate the build process in e² studio.
4. Identify the `<project_name>.bin` file located in the build directory, for example: `\e2_studio\workspace\<project_name>\Debug\<project_name>.bin`.
   Rename this file to `RA6Wx_cache.bin` as applicable.

5. Replace the previously added dummy file `RA6Wx_cache.bin` from the `SBOOT/image` directory with the newly created file, as shown in Figure 28.

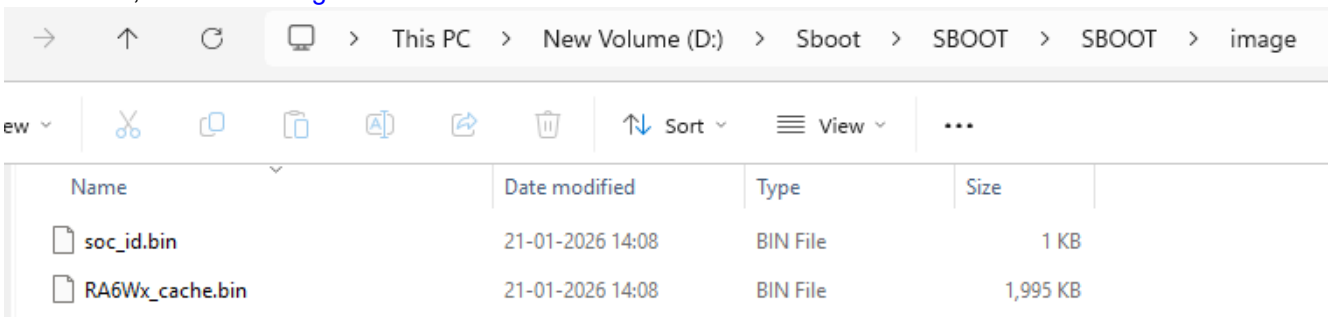

Figure 28. RA6Wx_cache.bin

4.5 Create Secure Boot Image

After you place the `RA6Wx_cache.bin` in the `/SBOOT/image` directory, create the secure image:

- In the SBOOT tool, click **Secure Boot** (Figure 29).

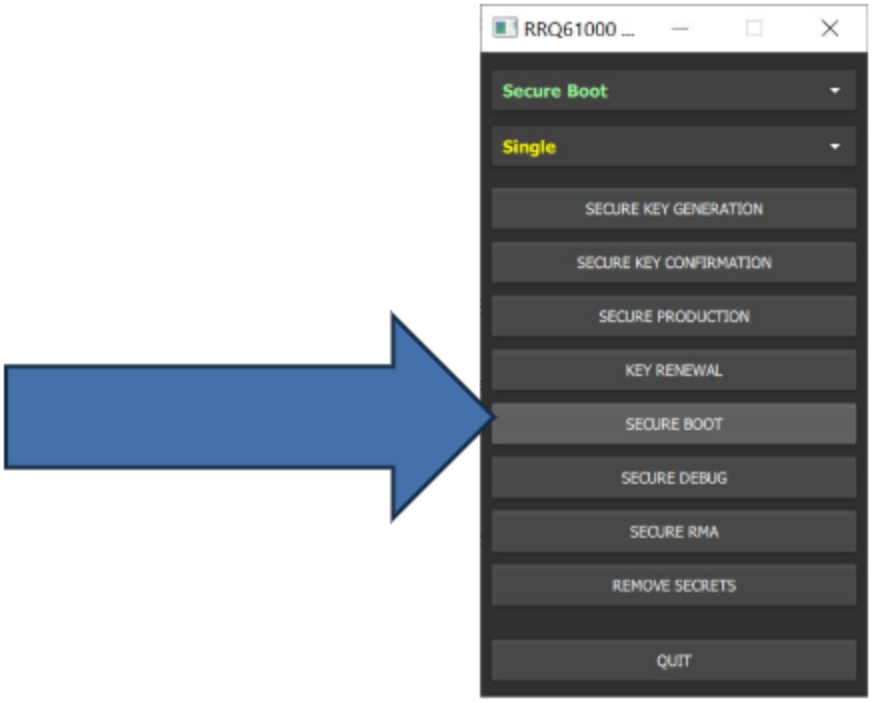

Figure 29. Secure Boot button

The `RA6Wx_RTOS.img` file is created in the `/SBOOT/public` directory.
Use this image to program the RA6W1 or RA6W2 device. After loading the image onto the board, the device enters Secure State. For more details and instructions on how to proceed, see Section 4.4 Secure Production.

4.6 Key Renewal

If either the second or the third private keys are compromised for any reason, replace these keys using the **Key Renewal** button in the main menu.
Use caution before selecting Key Renewal; activating this function deletes all previously generated second and third private keys and certificates and regenerate them from scratch.
Note that the Root of Trust (first private key) cannot be modified.

1. In the SBOOT tool, click **Key Renewal**.
2. When prompted, click **Yes to All** (Figure 30).

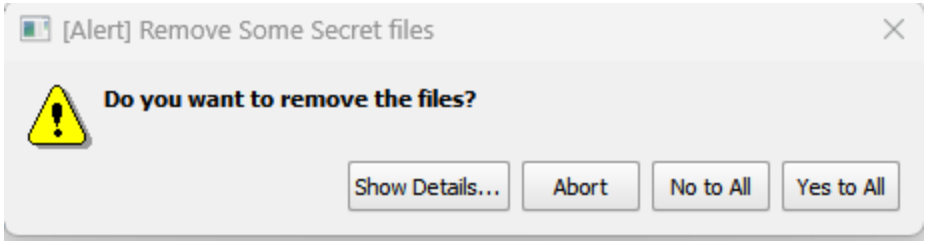

Figure 30. Remove Some Secret files alert

3. (Optional) If necessary, you can increment the anti-rollback counter (CM.NVCount), see Figure 31.

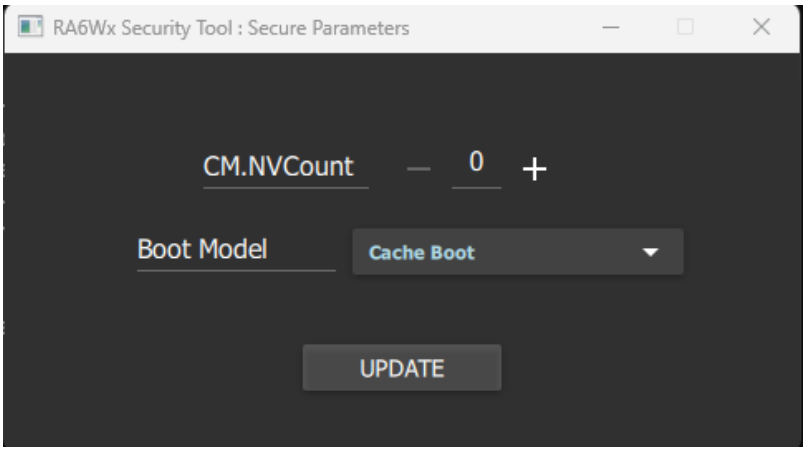

Figure 31. Secure key renewal CM NV count update

4. Repeat this procedure for the DM state, which appears sequentially. See Figure 33 related to the DM process. If necessary, you can update DM anti-rollback counter (Figure 32).

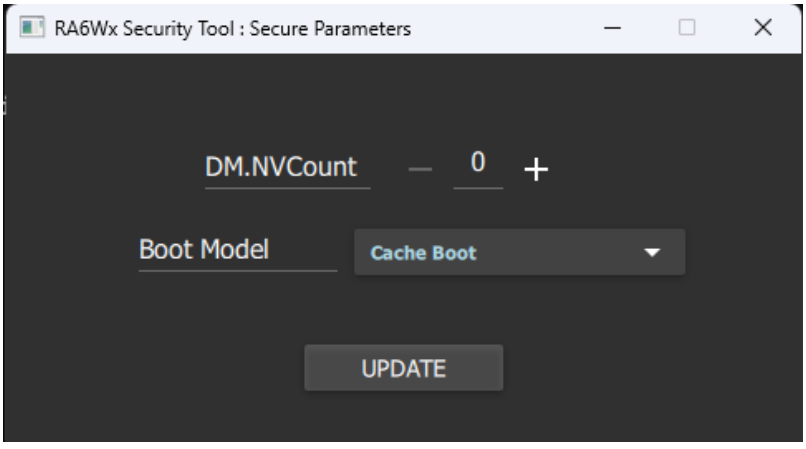

Figure 32. Secure key renewal DM NV count update

5. In the **RTOS.cache** window, from the **Encrypt Scheme** drop-down menu, select the **WKEY_CODE_ENC** option for DM image (Figure 33).

NOTE
For the RA6W2 platform, rename the COMP1 and IMAGE name from RA6W1 to RA6W2.

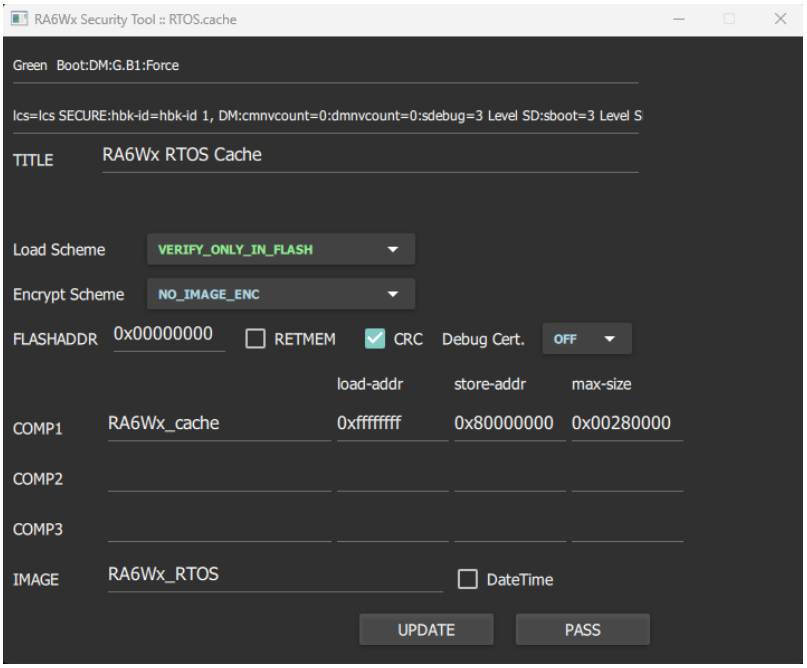

Figure 33. Key renewal DM config changes

After completing the steps, Secure Boot images incorporating the certificate chain based on the renewed keys are generated within the `/SBOOT/public` directory. The `key_renewal.txt` file, located in the `example` directory, serves as a log for the Key Renewal process and can be referenced to review logs or identify any error messages that occurred.

Table 11 summarizes which directories are updated after key renewal.

Table 11. Directory definition for key renewal

Directory	Contents
dmsecret	Second/third private keys for Secure Boot and Secure Debug.
dmpublic	First/Second/Third certificates for Secure Boot and Secure Debug that use Hbk1 (DM root key).
dmpubkey	Second/Third public keys.
dmtpmcfg	Configurations.
public	Images with the certificate chain for Secure Boot.

4.7 Secure Boot

1. In the SBOOT tool, click **Secure Boot**.
   Secure Boot images with the certificate chain are generated in the `/SBOOT/public` directory. See Section 4.5 Create Secure Boot Image.
   a. Save the `RA6Wx_RTOS.bin` and `RA6Wx_RTOS.img`.
   b. Verify that the `.bin` file hex content includes the version name in the header and is marked as "raw".

```
head -c 128 RA6Wx_RTOS.bin | hexdump -C
```

```
00000000  44 41 31 36 0d 11 39 59  52 41 36 57 31 2d 52 52  |DA16..9YRA6W1-RR|
00000010  51 36 31 30 30 31 2d 37  37 30 39 31 32 36 61 30  |Q61001-7709126a0|
00000020  31 2d 36 37 30 31 31 2d  63 6d 61 6b 65 00 00 00  |1-67011-cmake...|
00000030  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000040  00 00 00 00 00 00 00 00  00 18 00 00 60 da 20 00  |............`. .|
00000050  fd 84 e0 bf 72 61 77 6e  68 0a 00 00 1e ae 73 86  |....rawnh.....s.|
00000060  0e dc 3d 3b 63 6b 42 53  00 00 01 00 72 00 00 00  |..=;ckBS....r...|
00000070  01 00 00 00 ba ee c8 55  b1 a9 3b 6a a0 58 50 7d  |.......U..;j.XP}|
```

Figure 34. Verify header and version name

   c. Program the RTOS Image onto the RA6W1 board and, in the console, run the following commands:

```
sbrom
run cmpu
```

```
[/RRQ61000] #
[/RRQ61000] # sbrom
[/RRQ61000/sbrom] #
[/RRQ61000/sbrom] #
[/RRQ61000/sbrom] #
[/RRQ61000/sbrom] # help
sbrom commands:
        run             :  [dbg|secure|otp|ao|socid|asset|trng|suite-num] [iter]
        view            : view config
        set             : set config
        clear           : clear config

[/RRQ61000/sbrom] # run cmpu

 *=== SECURE PRODUCTION ==========================*
 * While running this,
 * the secret keys will be registered in Secure OTP.
 * Please note that these keys are irrevocable
 *   and can only be removed when you run RMA.
 *=== SECURE PRODUCTION ==========================*

Selected Item[0]
[0][thread_peripheral_sbrom]
CC312 SBROM Test Routine
SBROM lcs[00000000]
LifeCycle: CM
Security Lifecycle:CM
SecureProduction.CMPU-PATTERN Test @ lcs-CM
CMPU OK

* ------------------------------*
* Please Turn-OFF the system !!
* If you have completed it correctly, you should do POR-Reboot quickly
*  to keep your secret keys securely.
* ------------------------------*
sbrom-Rslt: GOOD

[/RRQ61000/sbrom] #
```

Figure 35. Run cmpu example

2. Power-on reset the board (the board is booted with a new lifecycle).

3. Verify the lifecycle state change in the console using the following command and verify that the lifecycle has changed to DM:

```
run socid
```

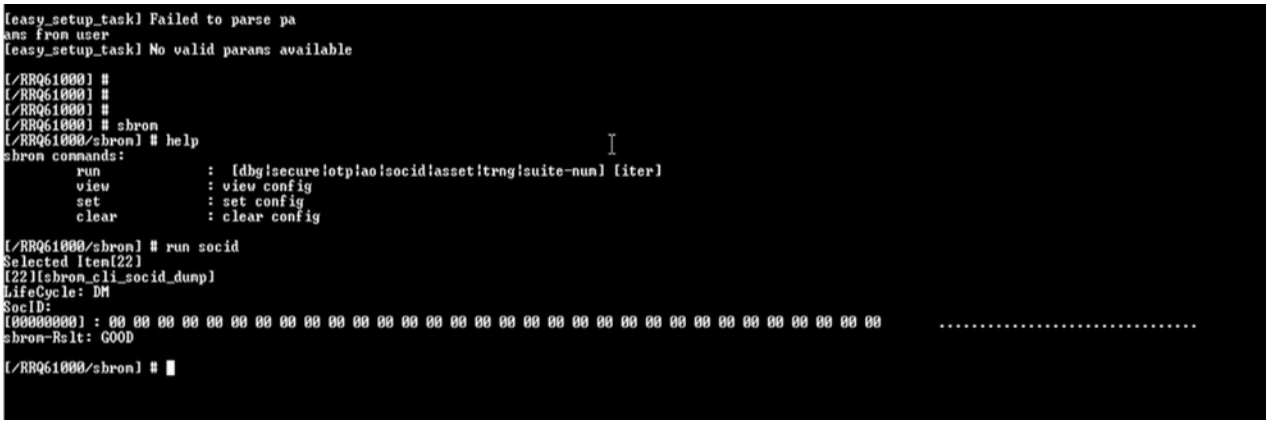

Figure 36. Run socid example

4. Run the `dmpu` transition command.

```
sbrom
run dmpu
```

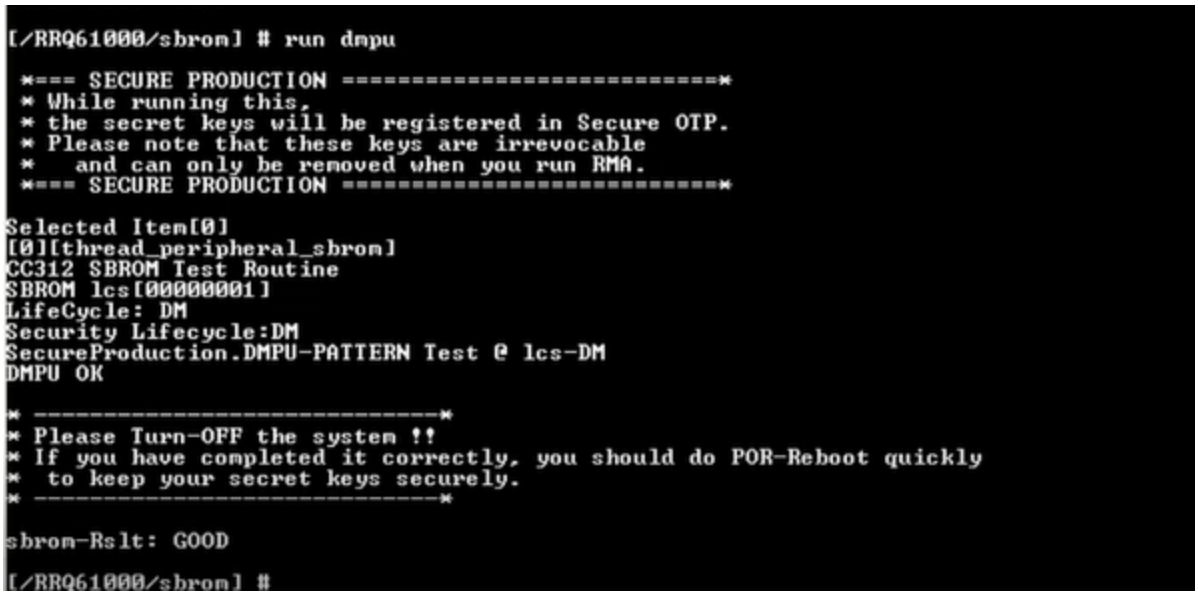

Figure 37. Run dmpu example

5. Power-on reset the board. When the board is rebooted, verify the LCS again.

```
sbrom
run dmpu
```

6. Verify that the device's LCS (Lifecycle State) has changed to SECURE, as shown in the Figure 38. Note that the Soc-ID field is also filled with values. After completing the previous steps, the board transitions to its Secure State.

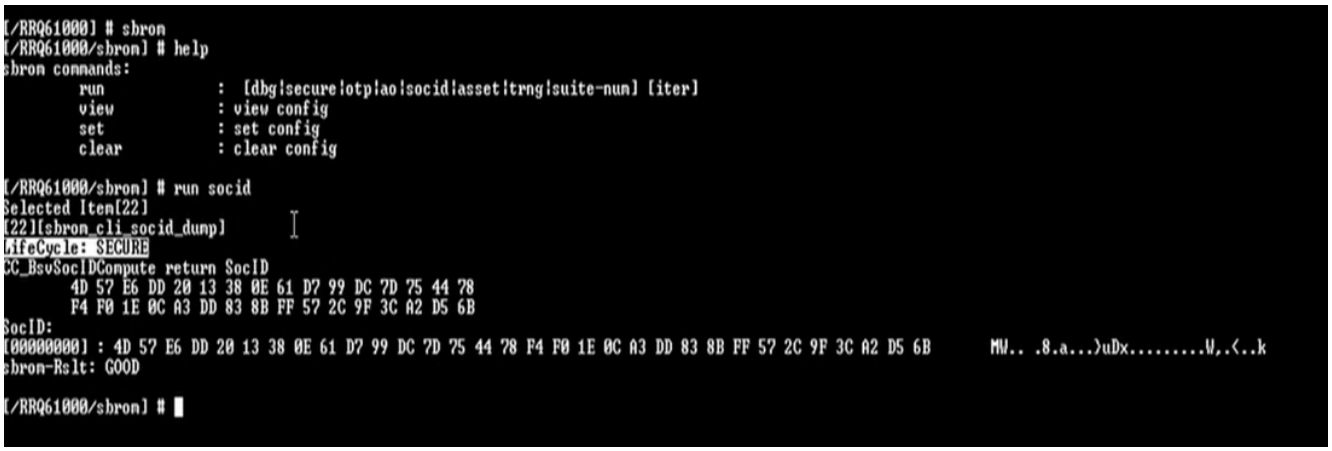

Figure 38. LCS check using SoC-ID command

4.8 Secure Debug

The debug port in the RA6W1/RA6W2 JTAG is disabled by default when entered Secure LCS. When this debug port needs to be re-enabled for debug purposes, then a Secure Debug image should be used. There is an optional Debug certificate field in an image.

At the boot sequence, a check is done to see whether the Debug certificate exists in the image. If a Debug certificate exists, then the SoC-ID in the Debug certificate is checked to confirm it matches the target device. When it does match, the debug port is enabled and booted. When you click **Secure Debug** in the SBOOT tool (Figure 39), the window shown in Figure 40 displays to enter the SoC-ID of the target device. Use the `lcs` command in the console to check what the SoC-ID is of the target device.

To use Secure Debug:

1. In the SBOOT tool, click **Secure Debug**.

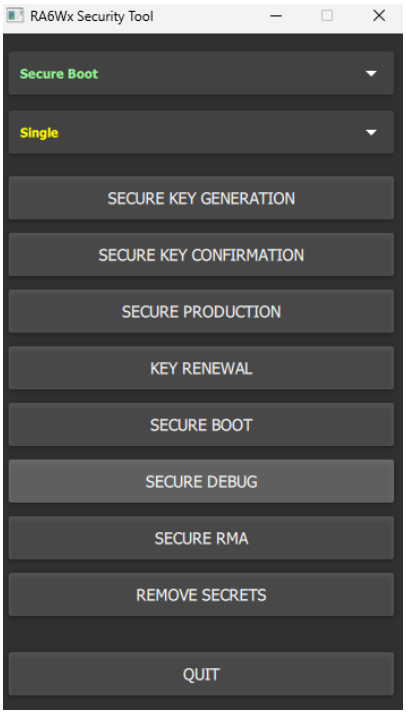

Figure 39. Secure Debug button

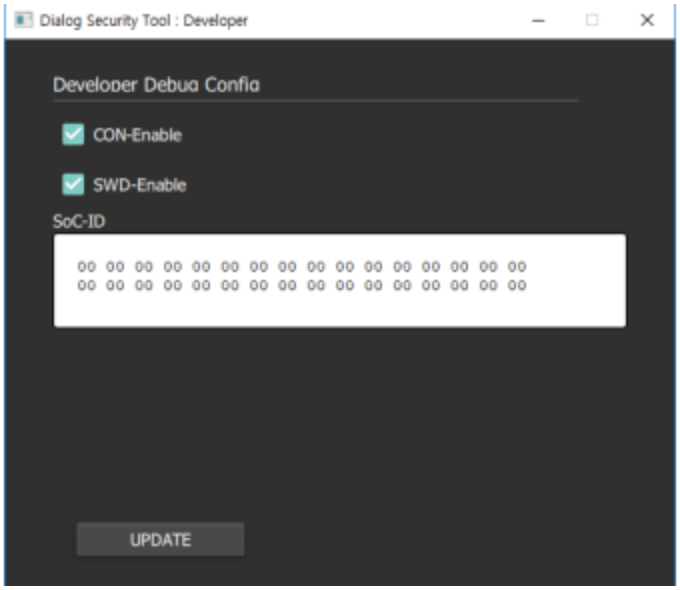

Figure 40. Secure debug SOC-ID not filled

2. You can configure the debug option as needed. To retrieve the SoC-ID, use the `lcs` command, see Figure 41 for guidance. When the RA6W1/RA6W2 boots up in a secure mode, it can print out its SoC-ID as shown in Figure 41.

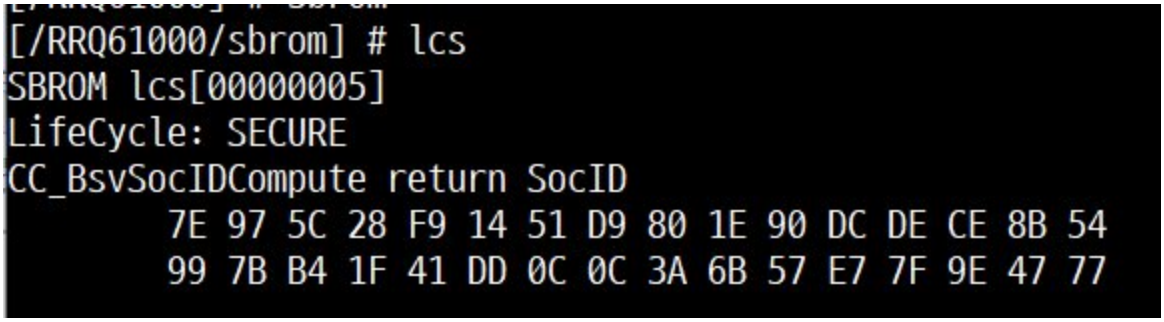

Figure 41. `lcs` command to get the SoC-ID

The image that matches SOC-ID can activate the debugger in the secure mode.

3. Copy the SoC-ID displayed in the console to the input field shown in Figure 42, and then click **Update** to continue.

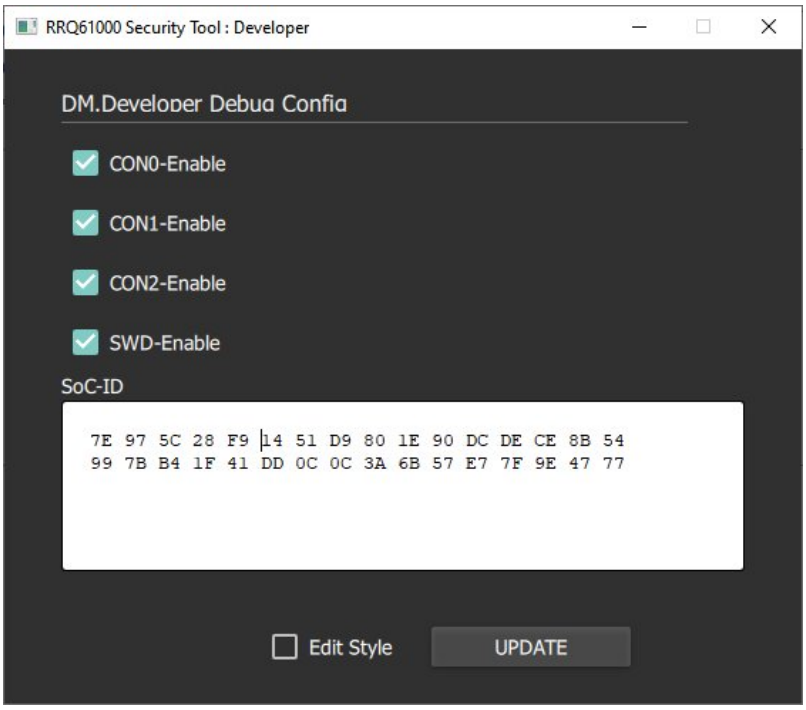

Figure 42. Fill the SoC-ID field and update

Table 12 summarizes which directories are updated from Secure Debug.

Table 12. Directory definition for secure debug

Directory	Contents
dmpublic	Developer certificate with the SoC-ID.
public	Images with Debug certificate.

The `RRQ61000_RTOS.img` file, generated in the `/SBOOT/public` directory, includes the secure certificate required for enabling secure debug functionality.
The `secure_debug.txt` in the `example` directory is a log file for the Secure Debug process. The file can be used to check the log and read error messages that occurred.

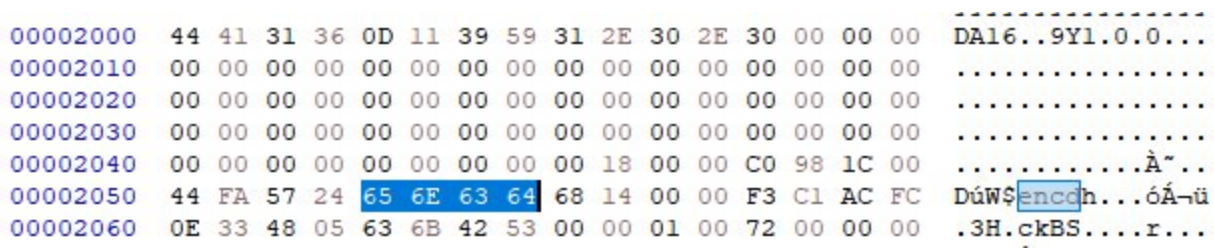

Figure 43. Example of the debug image

When the RA6W1/RA6W2 boots up with secure image, it can attach the debugger like J-Link.

4.9 Return Merchandise Authorization

The LCS of the chip should be changed to RMA-LCS before the chip is sent to the chip maker (for example, Renesas Electronics) for analysis.
A Debug certificate that has a Return Merchandise Authorization (RMA) flag enabled (RMA certificate) is required to enter a device into RMA LCS. Like Secure Debug, Secure RMA is allowed for a specific device, and a SoC-ID is required for the RMA certificate.
When changing to RMA-LCS, secret keys in the OTP memory such as Kpicv, Kceicv, Kcp, and Kce are erased to prevent that the developer's secret keys are exposed, and the debug port (JTAG) is re-enabled for debugging purposes.
After this, Renesas can debug it with a non-secure image which does not require secret keys in OTP memory. Erasing secret keys and data in OTP memory can be done with a boot image with the authorized Debug certificate with the enabled RMA flag.
There are two steps for the RMA:

- DM RMA
- CM RMA

To use Secure RMA, complete the following:

1. In the SBOOT tool, click **Secure RMA** (Figure 44).

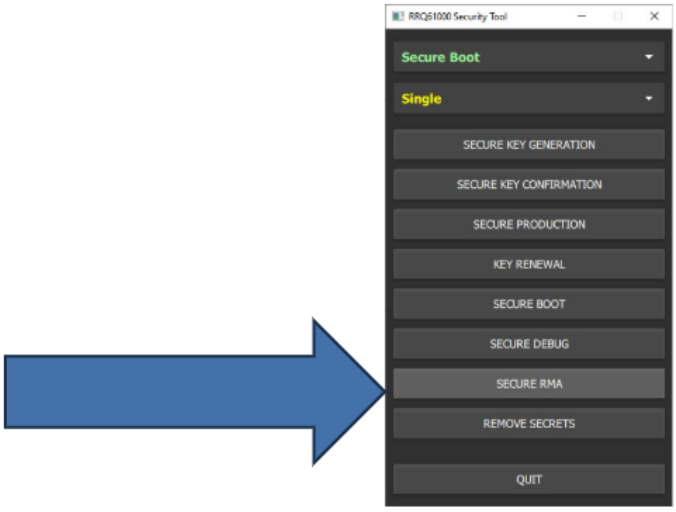

Figure 44. Secure RMA button

2. In the **Enabler RMA** window (Figure 45), click **Pass**.

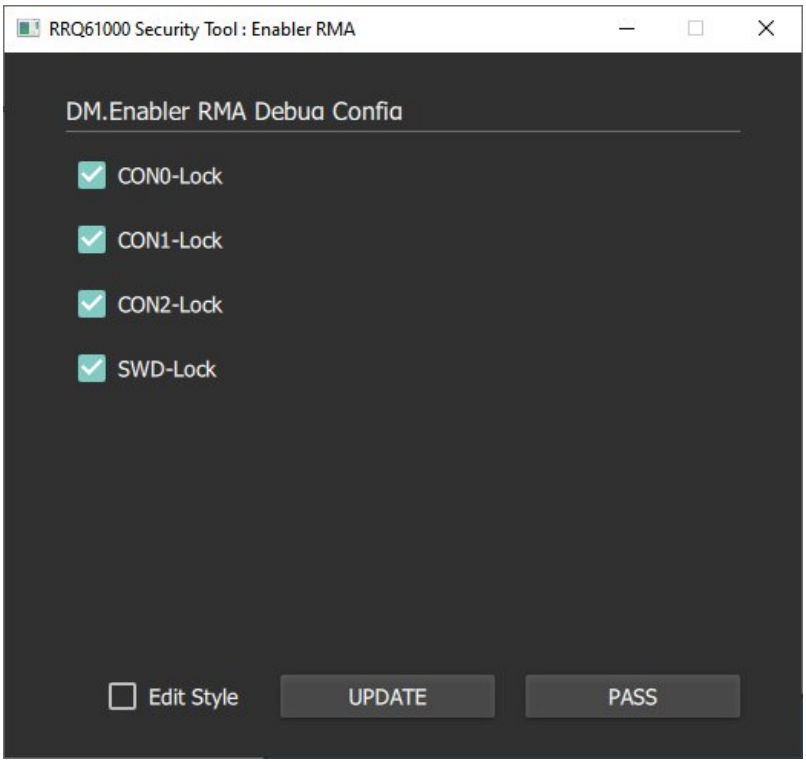

Figure 45. DM RMA window

3. In the **Developer RMA** window (Figure 46), enter the SoC-ID, and then click **Update**.

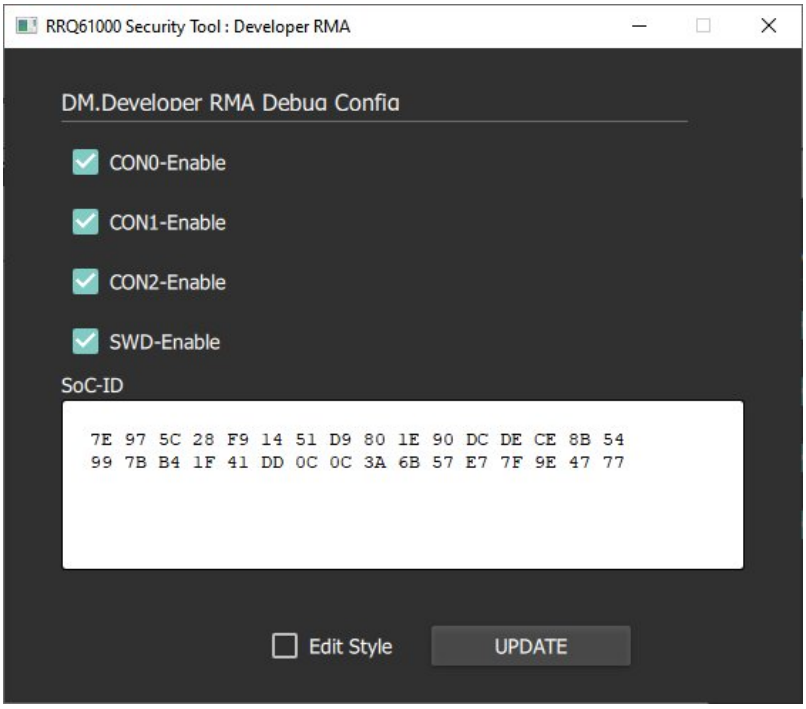

Figure 46. DM RMA SoC-ID entry

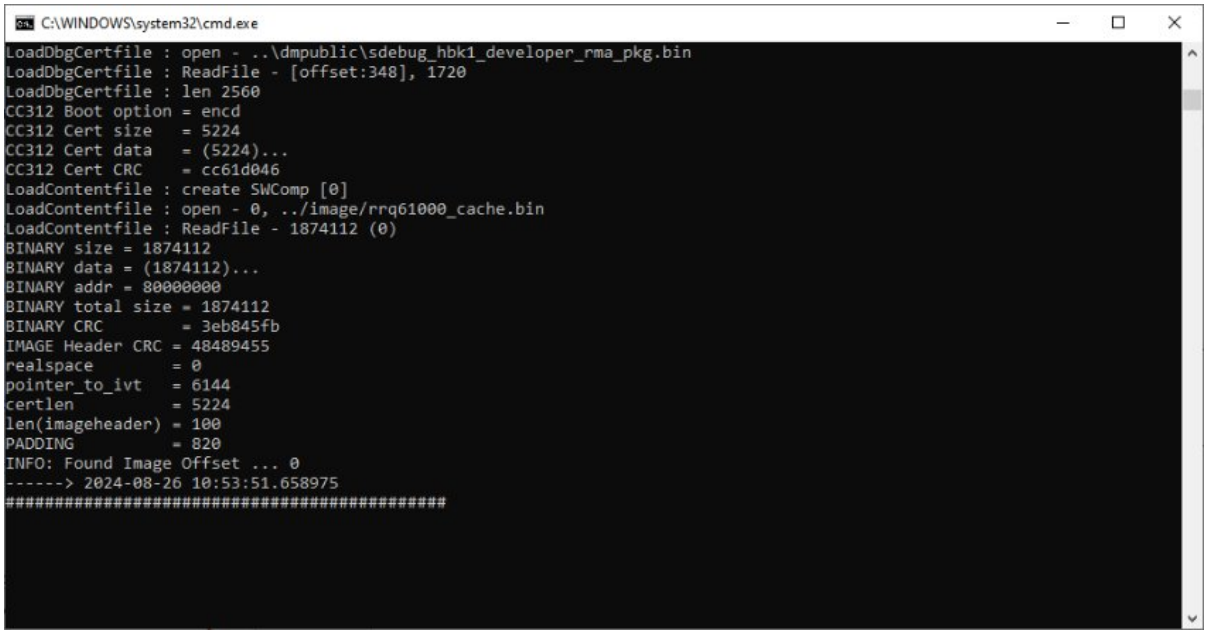

Figure 47. DM RMA completion console log

When the DM RMA is completed (Figure 47), it proceeds with the CM RMA procedure to familiarize with CM RMA:

1. In the **Enabler RMA** window (Figure 48), click **Update**.

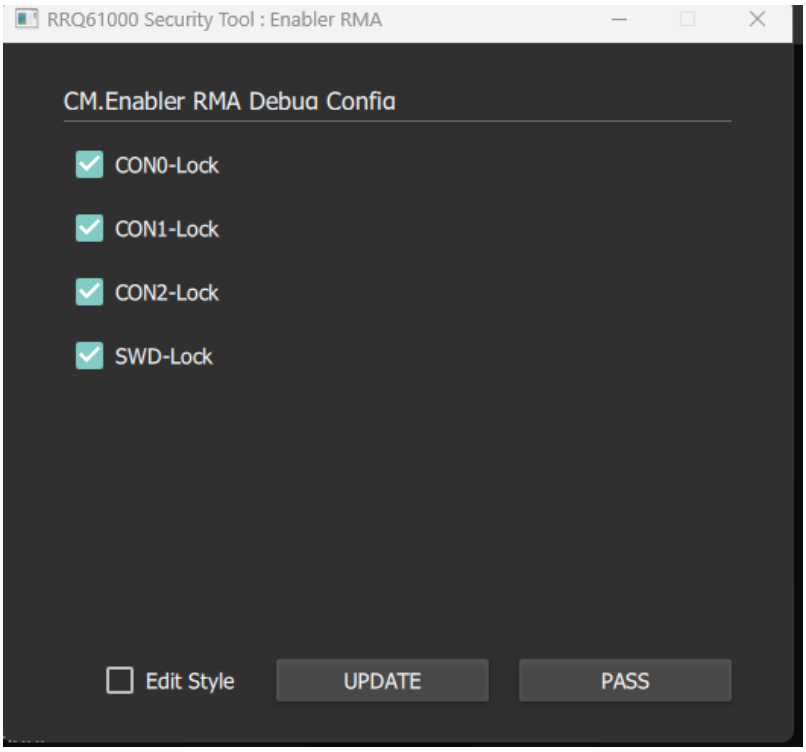

Figure 48. CM RMA

2. In the **Developer RMA** window (Figure 49), click **Update** again.

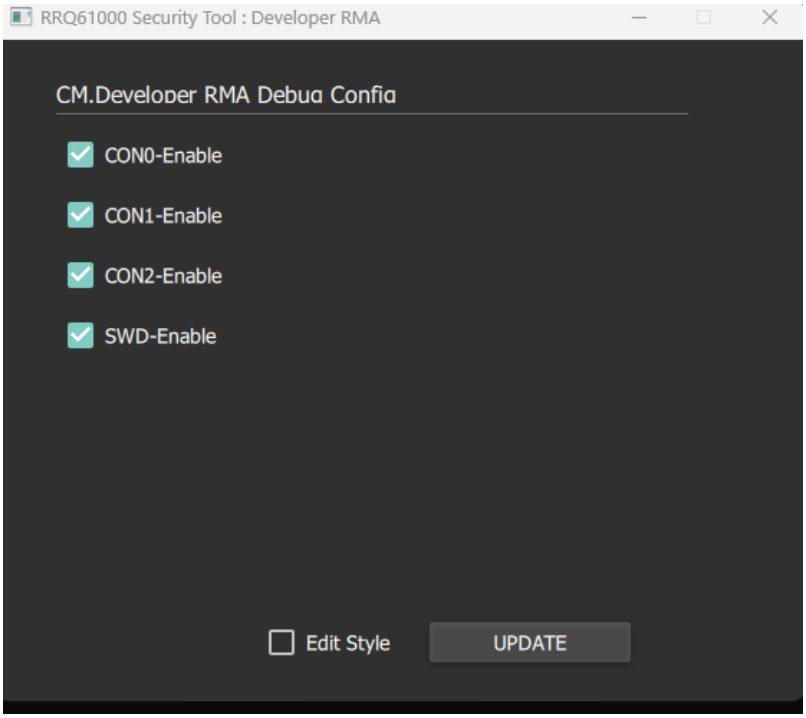

Figure 49. CM RMA developer config

3. After completing the steps, verify the Console log output to confirm the successful RMA procedure as shown in Figure 50.

```
BOOT model 0
INFO: Found Image Offset ... 2000
ALIGN image      = 4065
LoadCertfile : create certchain [0]
LoadCertfile : open - 0, ../cmpublic/sboot_hbk0_3lvl_key_chain_issuer.bin
LoadCertfile : ReadFile - 840 (0)
LoadCertfile : create certchain [1]
LoadCertfile : open - 1, ../cmpublic/sboot_hbk0_3lvl_key_chain_publisher.bin
LoadCertfile : ReadFile - 840 (1)
LoadCertfile : create certchain [2]
LoadCertfile : open - 2, ../cmpublic/sboot_hbk0_rma_cert.bin
LoadCertfile : ReadFile - 868 (2)
LoadDbg.Key.Certfile : open - ..\cmpublic\sdebug_hbk0_3lvl_key_chain_enabler.bin
LoadDbg.Key.Certfile : ReadFile - 840
LoadDbgCertfile : open - ..\cmpublic\sdebug_hbk0_developer_rma_pkg.bin
LoadDbgCertfile : ReadFile - [offset:348], 1720
LoadDbgCertfile : len 2560
CC312 Boot option = rawd
CC312 Cert size   = 5224
CC312 Cert data   = (5224)...
CC312 Cert CRC    = 661f1d2f
LoadContentfile : create SWComp [0]
LoadContentfile : open - 0, ../image/rrq61000_cache.bin
LoadContentfile : ReadFile - 2060244 (0)
BINARY size = 2060256
BINARY data = (2060256)...
BINARY addr = 80000000
BINARY total size = 2060256
BINARY CRC        = c0531a99
IMAGE Header CRC = 4916186c
realspace         = 0
pointer_to_ivt    = 6144
certlen           = 5224
len(imageheader) = 100
PADDING           = 820
INFO: Found Image Offset ... 0
------> 2025-05-22 19:17:46.353078
##############################################



======> Procedure has been completed successfully ...
```

Figure 50. CM RMA successful completion

4.10 Remove Secrets

When you want to have a third party (or developer) to debug the end-product in the field, you need to use **Remove Secrets** before the SBOOT directory is delivered to the third party, to remove all important secret keys and certificates. Before clicking the **Remove Secrets** button, the original SBOOT directory should be already backed up in a safe location because all secret keys are removed. Then, the third party can make its own debug images with the SBOOT environment.

After debugging is complete by the third party, you should apply the resolving patch codes from the third party to the SDK and build the SDK, which makes RTOS binary in SDK that are copied to the `image` directory.

1. In the SBOOT tool, click **Remove Secrets**.
2. When prompted, to remove all secrets, click **Yes to All** (Figure 51).

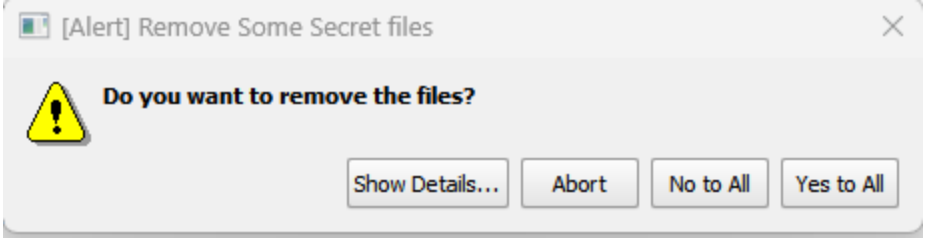

Figure 51. Prevent accidental removal of secret keys in Secure RMA

3. Click the needed target publisher that should send the `SBOOT` directory and what files should be removed accordingly, see Figure 52.

NOTE
Table 13 summarizes the files to be removed based on the selected target.

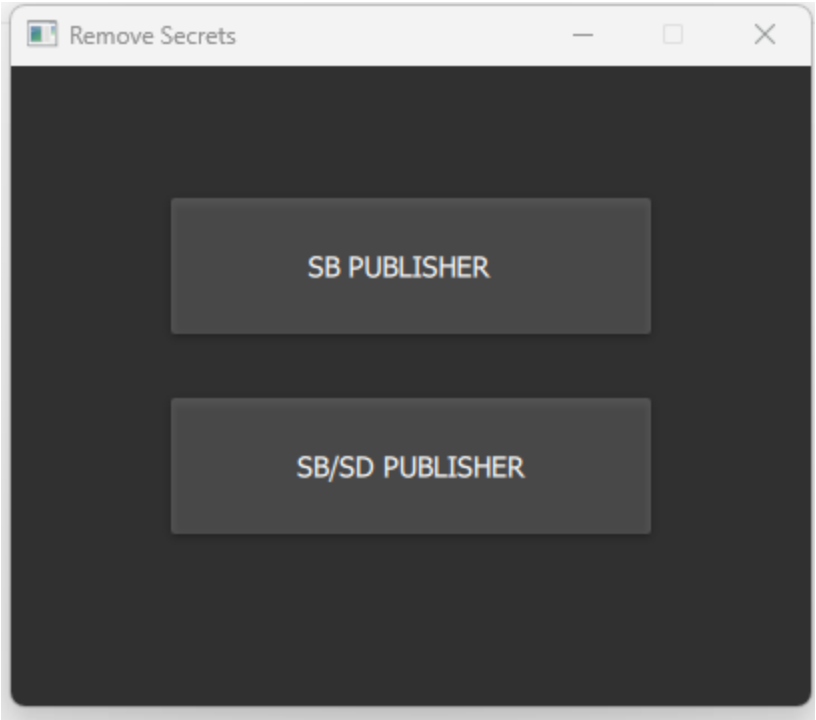

Figure 52. Remove secret keys in Secure RMA

Table 13. Directory definition to remove secret keys in secure RMA

Target	Directory	Removed files
SB Publisher	cmsecret	All files
	cmpublic	All files
	dmsecret	All files except dmpublisher_keypair.pem
	dmpublic	enc.kce.bin, enc.kcp.bin, and all sdebug_* files
SB/SD Publisher	cmsecret	All files
	cmpublic	All files
	dmsecret	All files except dmpublisher_keypair.pem, dmdeveloper_keypair.pem
	dmpublic	enc.kce.bin, enc.kcp.bin, sdebug_hbk1_enabler_rma_pkg.bin, sdebug_hbk1_developer_rma_pkg.bin

After this, the `SBOOT` directory can be sent to the third party (or developer) for debugging or development purposes.

Appendix A Secure AT Channel and Secure Key Provisioning

This section describes the procedure for preparing the groundwork and establishing a secure AT communication channel using the AT secure channel key and encrypted AT commands transmitted from the host to the RA6W. The process begins by generating a secure asset from the AT Secure Channel Key, using the SBOOT tool, before programming this secure asset to the designated AT Secure Channel Key address in the SFLASH.

NOTE
The RA6W1 must use a Secure Boot image to enable AT Secure Channel and Secure Key Provisioning, as the Hardware Key (Kcp) used to encrypt the keys must be provisioned to the device as part of the Secure Boot image.

A.1 Generate and Flash the AT Secure Channel Key to RA6W1 as a Secure Asset

To generate a secure asset from the AT secure channel key using the SBOOT tool:

1. Download and extract the SBOOT tool on the host PC.
2. Using a software for ASCII-to-HEX conversion (such as HxD Freeware Hex Editor), create a new file and enter your desired secure channel key (also referred to as the "known key" in this document) for the AT Secure Channel as decoded/plain text.

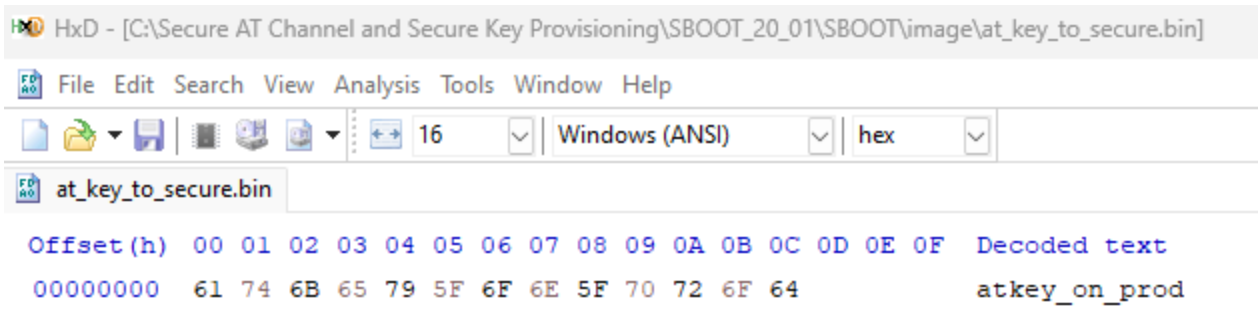

Figure 53. Provide the known key

In this example, the known key **atkey_on_prod** is used.
The length of the key must be 16 bytes. If the key comes up short (such as in example), you can add groupings of 0x00 (NULL) in the Hex editor to compensate.

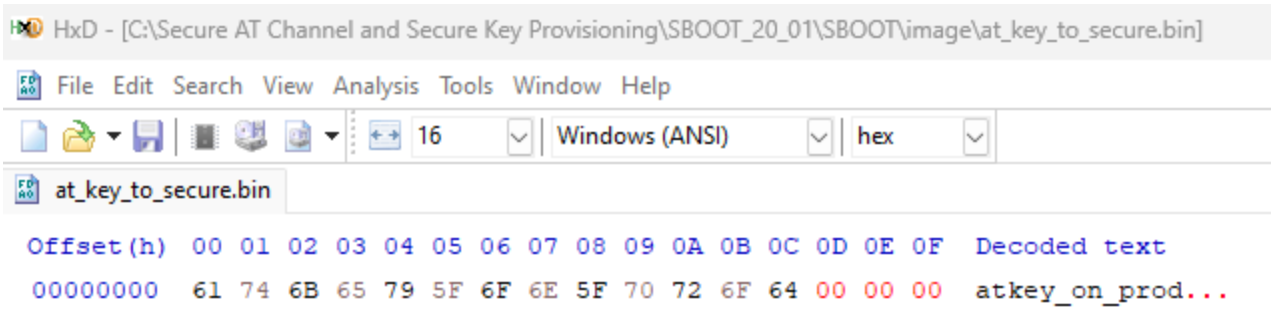

Figure 54. Example key

3. Save the file as `at_key_to_secure.bin` in the `/SBOOT/image` directory.

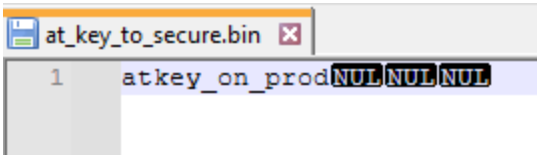

Figure 55. Example at_key_to_secure

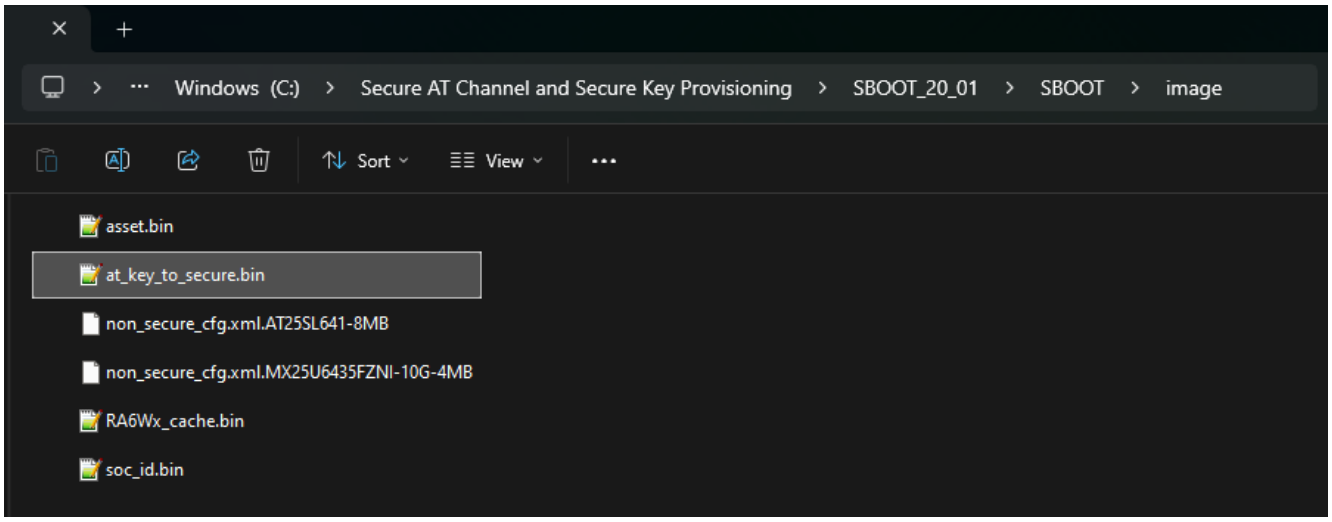

Figure 56. at_key_to_secure in `Image` directory

This key is converted into a Secure Asset package by the SBOOT tool, which will subsequently be provisioned to the SFlash on the RA6W1 and used as the AT Secure Channel key.

4. Run `CM.4.secuasset.bat` in the `SBOOT` directory and provide the path to your input file (`at_key_to_secure.bin`) and output file (`secure_asset.pkg.bin`), before clicking **Generate** to generate the Secure Asset.

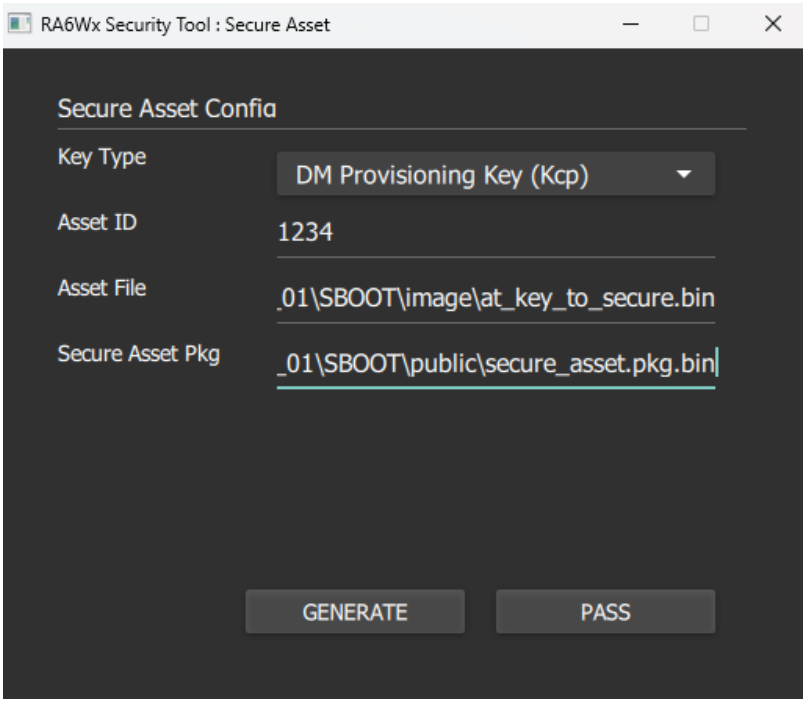

Figure 57. Generate secure asset

After the key is generated, you can see the `secure_asset.pkg.bin` in the `/SBOOT/public` directory (or your chosen location), as well as `secure_asset.pkg.txt`.

5. Using the `cli_programmer.exe` tool (over J-Link GDB Server), follow the below steps to write the secure key to the AT Secure Channel Key address in the RA6W1 SFlash:

```
cli_programmer.exe gdbserver erase_qspi 0x3fe080 0x400
cli_programmer.exe gdbserver  write_qspi 0x3fe080 secure_asset.pkg.bin[
```

6. Verify that the secure asset package has been successfully flashed by reading the secure memory using `cli_programmer.exe`:

```
cli_programmer.exe gdbserver read_qspi 0x3fe080 -- 0x100
```

```
D:\Sboot\SBOOT\SBOOT\public>cli_programmer.exe gdbserver read_qspi 0x3fe080 -- 0x100
cli_programmer 1.27
Copyright (C) 2023-2024 Renesas Electronics

bootloader file not specified, using internal uartboot.bin

Uploading boot loader/application executable...
chip_rev is 400AA
Executable uploaded.

003FE080   74 65 73 41 00 00 01 00 40 00 00 00 00 00 00 00 00 00 00 00 8E AC D0 62 79 E5 E2 11 47 D2 59 29   tesA....@..
...........by...G.Y)
003FE0A0   CC E8 0B 89 43 17 55 2B 64 45 3B 47 E9 4F EE B1 77 81 2D EC 84 FE 8F E5 51 48 05 8A 84 4C 9C E7   ....C.U+dE;
G.O..w.-.....QH...L..
003FE0C0   AB 1C 48 24 C3 BB 57 B3 AA CB FB B6 DC 5D BC DF 7C B8 FD A8 1E 08 5B 07 31 23 63 CB 58 7C FA E6   ..H$..W....
..]..|.....[.1#c.X|..
003FE0E0   B0 DD E6 D7 37 78 A5 8B 77 39 20 EE 17 F0 B0 DF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF   ....7x..w9
.....................
003FE100   FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF   ..........
.....................
003FE120   FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF   ..........
.....................
003FE140   FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF   ..........
.....................
003FE160   FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF   ..........
.....................
done.
```

Figure 58. Verify successful write operation in SFlash

A.2 Prepare and Flash the Secure Boot Image to the Device

To unpack the Secure Asset on the device, the Kcp that was used to encrypt it must be provisioned to the OTP of the RA6W1, and this is done as part of the Secure Boot Image provisioning.
To proceed, follow the Secure Boot instructions in Section 4.7 Secure Boot.

NOTE
This step must only be performed after provisioning the AT Secure Channel Key as a Secure Asset to the SLFASH on the RA6W1, as described in Appendix A.1 Generate and Flash the AT Secure Channel Key to RA6W1 as a Secure Asset.

A.3 Verifying the AT Secure Channel

AT Secure Channel requires synchronization between the Host PC and the target device (RA6W1).
Table 14 lists the implementation examples.

Table 14. Implementation examples

Device	Application	Description
Host PC	secure_channel.py	Python GUI Application
RA6W1	rm_wifi_test_app_sec_at_uart_ek_ra6w1_ep	Example e^2 studio application, to be built on top of FSP Pack. Demonstrates application layer implementation of AT Secure Channel on the ra6w1. Available in the RA_Solutions_con/example_projects/ek_ra6w1 Gitlab repository.

1. Program the RA6W1 device with an image built with the **rm_wifi_test_app_sec_at_uart_ek_ra6w1_ep** example template. See *RA6W1 Getting Started Guide* for further instructions on flashing an image to the device.
2. Open `secure_channel_v1.py` in a python editor and configure the `known_key` variable:
   In the example provided in Appendix A.1 Generate and Flash the AT Secure Channel Key to RA6W1 as a Secure Asset, whereby the AT Key in ASCII is `atkey_on_prod[NUL] [NUL] [NUL]`, the equivalent HEX value of this key is **61746b65795f6f6e5f70726f000000**. The `known_key` variable in the `secure_channel.py` program running on the host PC should receive this same hexadecimal value.
   For ease of use, the `secure_channel.py` program is designed to take the ASCII value of the key (for example, **atkey_on_prod**) and convert it to HEX, with the appropriate zero-padding, automatically.

```
# Status variables
secure_channel_active = False
known_key = "atkey_on_prod".encode("ascii").ljust(16, b"\x00") # AES-128 key: ASCII + zero padding
random_iv = get_random_bytes(16)
last_encrypted_block = random_iv
last_received_block = random_iv
stop_receiving = False
```

Figure 59. Known key

3. Run `secure_channel.py` on the host (pyserial and pycryptodome library imports are required).
4. From the GUI, select the AT console COM port of the device, and then click **Open Port**.

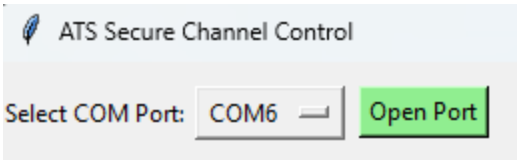

Figure 60. Select COM Port

5. Click **Enable Secure Channel**. Do not manually enter an IV value, it is randomly generated.

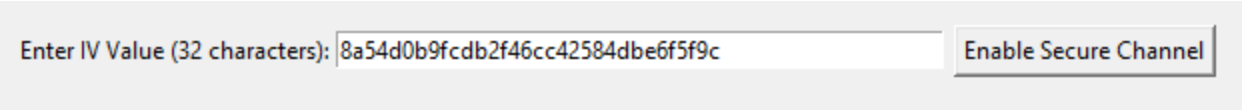

Figure 61. Enable Secure Channel

After activation, the GUI plain data log should display "Secure Channel enabled", and all transmitted AT commands will be encrypted using the AT key.

6. In the **Enter full AT Command (including arguments)** field, enter AT commands. The GUI displays both the encrypted and the plain-text logs or verification.

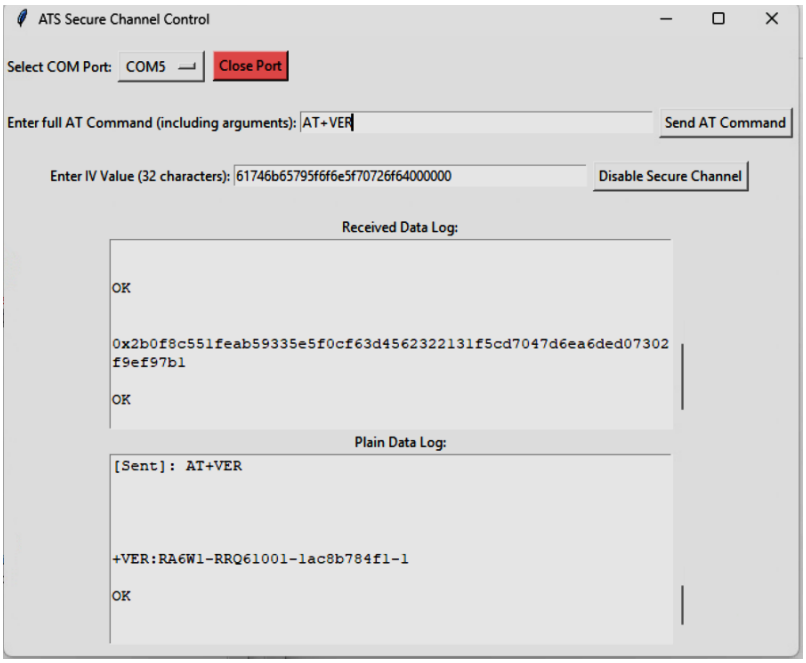

Figure 62. AT secure channel

A.4 Secure Key Provisioning with AT Commands and Secure AT Channel

Secure Key Provisioning refers to the process of securely storing cryptographic keys within the device, ensuring they are protected from unauthorized access or tampering. The Secure AT Channel enables encrypted communication between the host and the device for AT commands, enhancing data confidentiality during transmission. These are two agnostic features and can operate independently: when provisioning keys with the ESC<CERT> AT Command and a Secure Boot image, the keys shall remain securely provisioned and protected in storage, regardless of whether the Secure AT Channel is active.

Upon meeting the covered requirements for AT Secure channel, certificates may also be provisioned over the secure AT channel by using the dedicated <ESC>CERT AT commands. See *RA6W1 and RA6W2 AT Commands*, *<ESC>CERT commands* for usage instructions.

For example, to provision the AWS Root CA:

1. Write the <ESC>CERT command, following the syntax shown in the example:

```
\x1BCERT,6,0,0,1,1187,-----BEGIN CERTIFICATE-----
MIIDQTCCAimgAwIBAgITBmyfz5m/jAo54vB4ikPmljZbyjANBgkqhkiG9w0BAQsF
ADA5MQswCQYDVQQGEwJVUzEPMA0GA1UEChMGQW1hem9uMRkwFwYDVQQDExBBbWF6
b24gUm9vdCBDQSAxMB4XDTE1MDUyNjAwMDAwMFoXDTM4MDExNzAwMDAwMFowOTEL
MAkGA1UEBhMCVVMxDzANBgNVBAoTBkFtYXpvbjEZMBcGA1UEAxMQQW1hem9uIFJv
b3QgQ0EgMTCCASIwDQYJKoZIhvcNAQEBBQADggEPADCCAQoCggEBALJ4gHHKeNXj
ca9HgFB0fW7Y14h29Jlo91ghYPl0hAEvrAIthtOgQ3pOsqTQNroBvo3bSMgHFzZM
9O6II8c+6zf1tRn4SWiw3te5djgdYZ6k/oI2peVKVuRF4fn9tBb6dNqcmzU5L/qw
IFAGbHrQgLKm+a/sRxmPUDgH3KKHOVj4utWp+UhnMJbulHheb4mjUcAwhmahRWa6
VOujw5H5SNz/0egwLX0tdHA114gk957EWW67c4cX8jJGKLhD+rcdqsq08p8kDi1L
93FcXmn/6pUCyziKrlA4b9v7LWIbxcceVOF34GfID5yHI9Y/QCB/IIDEgEw+OyQm
jgSubJrIqg0CAwEAAaNCMEAwDwYDVR0TAQH/BAUwAwEB/zAOBgNVHQ8BAf8EBAMC
AYYwHQYDVR0OBBYEFIQYzIU07LwMlJQuCFmcx7IQTgoIMA0GCSqGSIb3DQEBCwUA
A4IBAQCY8jdaQZChGsV2USggNiMOruYou6r4lK5IpDB/G/wkjUu0yKGX9rbxenDI
U5PMCCjjmCXPI6T53iHTfIUJrU6adTrCC2qJeHZERxhlbI1Bjjt/msv0tadQ1wUs
N+gDS63pYaACbvXy8MWy7Vu33PqUXHeeE6V/Uq2V8viTO96LXFvKWlJbYK8U90vv
o/ufQJVtMVT8QtPHRh8jrdkPSHCa2XV4cdFyQzR1bldZwgJcJmApzyMZFo6IQ6XU
5MsI+yMRQ+hDKXJioaldXgjUkK642M4UwtBV8ob2xJNDd2ZhwLnoQdeXeGADbkpy
rqXRfboQnoZsG4q5WTP468SQvvG5
-----END CERTIFICATE-----
```

NOTE
\x1BCERT represents the <ESC>CERT command.

2. In the ATS Secure Channel Control GUI, copy the command to the **Enter full AT Command (including arguments)** field, and then click **Send AT Command**. You should see "OK" under **Received Data Log**, as well as see the actual **Plain Data Log**, as shown in Figure 63.

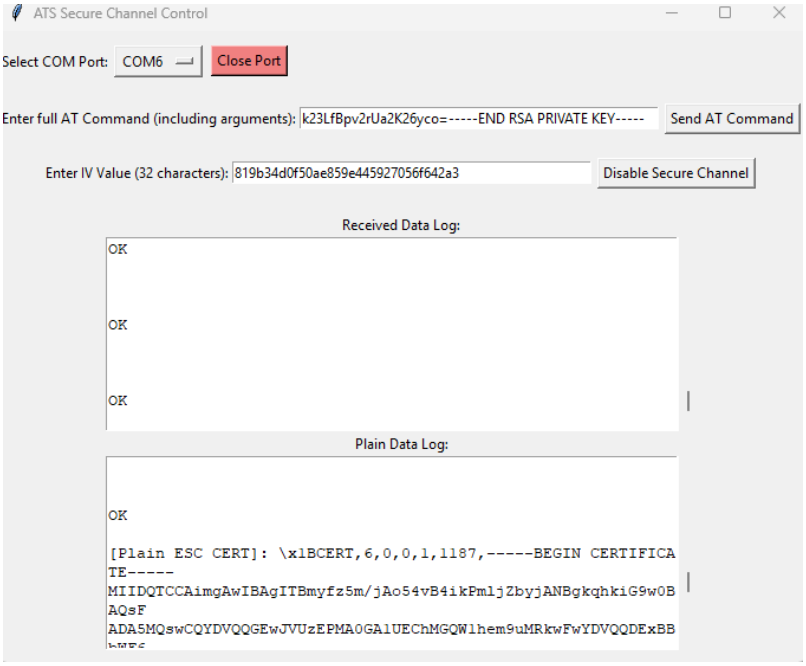

Figure 63. Esc Cert command over AT Secure

When provisioning using this method, the AT commands are encrypted between devices and the AT Secure Channel key is stored in a secured manner in the SFLASH, in accordance with the Secure Storage specification.

5. Revision History

Revision	Date	Description
1.00	June 10, 2026	First release.

IMPORTANT NOTICE AND DISCLAIMER

RENESAS ELECTRONICS CORPORATION AND ITS SUBSIDIARIES (“RENESAS”) PROVIDES TECHNICAL SPECIFICATIONS AND RELIABILITY DATA (INCLUDING DATASHEETS), DESIGN RESOURCES (INCLUDING REFERENCE DESIGNS), APPLICATION OR OTHER DESIGN ADVICE, WEB TOOLS, SAFETY INFORMATION, AND OTHER RESOURCES “AS IS” AND WITH ALL FAULTS, AND DISCLAIMS ALL WARRANTIES, EXPRESS OR IMPLIED, INCLUDING, WITHOUT LIMITATION, ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT OF THIRD-PARTY INTELLECTUAL PROPERTY RIGHTS.

These resources are intended for developers who are designing with Renesas products. You are solely responsible for (1) selecting the appropriate products for your application, (2) designing, validating, and testing your application, and (3) ensuring your application meets applicable standards, and any other safety, security, or other requirements. These resources are subject to change without notice. Renesas grants you permission to use these resources only to develop an application that uses Renesas products. Other reproduction or use of these resources is strictly prohibited. No license is granted to any other Renesas intellectual property or to any third-party intellectual property. Renesas disclaims responsibility for, and you will fully indemnify Renesas and its representatives against, any claims, damages, costs, losses, or liabilities arising from your use of these resources. Renesas' products are provided only subject to Renesas' Terms and Conditions of Sale or other applicable terms agreed to in writing. No use of any Renesas resources expands or otherwise alters any applicable warranties or warranty disclaimers for these products.

(Disclaimer Rev.1.01)

CorporateHeadquarters

TOYOSU FORESIA, 3-2-24 Toyosu
Koto-ku, Tokyo 135-0061, Japan

www.renesas.com

Trademarks

Renesas and the Renesas logo are trademarks of Renesas Electronics Corporation. All trademarks and registered trademarks are the property of their respective owners.

Contact Information

For further information on a product, technology, the most up-to-date version of a document, or your nearest sales office, please visit www.renesas.com/contact-us/