UG10106

Code Signing Tool User Guide

Rev. 4.0 — 5 December 2024 **User guide**

Document information

Information	Content
Keywords	UG10106, code signing, CST, HABv4, AHAB, SRK, CSF, PQC, i.MX
Abstract	The Code Signing Tool provides support for signing and encrypting images to be used with HAB and AHAB enabled NXP processors.

1 About this document

This document provides details related to installing, configuring, and running the code signing tool (CST).

Audience

This document is intended for administrators and engineers performing code signing for the NXP High Assurance Boot (HAB) and Advanced High Assurance Boot (AHAB) features.

Scope

This document explains how to use the CST to generate keys, certificates, HABv4/AHAB SRK tables, SRK hash values, and data that includes digital signatures. Loading images and burning eFuses using the NXP manufacturing tool are outside the scope of this document.

Organization

The remainder of this document is divided into sections according to the main HAB code signing tool user tasks:

- Section 2 "Introduction": Describes the background of the code signing tool (CST) and the goals of the procedures in later sections.
- Section 3 "Installation": Explains how to install the CST program files.
- Section 4 "Key and certificate generation": Explains how to generate signing keys and certificates for the HAB Version 4 and AHAB.
- Section 5 "CST usage": Explains how to use the CST client command-line interface.
- Section 6 "CSF description language": Provides details about the CSF description language and explains how to create a CSF description file.
- Section 7 "CST architecture": Provides details about CST implementation and customization options.

Conventions

This document uses the following notational conventions:

- `Courier New`: Indicates commands, command parameters, code examples, expressions, data types, directives, and file names.
- *Italic*: Indicates replaceable command parameters.
- All source code examples are in the C language.

All information provided in this document is subject to legal disclaimers.

© 2024 NXP B.V. All rights reserved.

2 Introduction

The code signing tool provides support for signing and encrypting images to be used with HAB and AHAB enabled NXP processors.

2.1 Code signing components

Many NXP processors include the secure boot feature with HAB or AHAB. The secure boot feature is based on public key infrastructure (PKI). The secure systems consist of two main components:

- The HAB library subcomponent of NXP processor boot ROMs or the AHAB secure subsystem that includes a dedicated Arm core, ROM, and firmware
- The CST

2.1.1 Secure components

The HAB library is a subcomponent of the boot ROM and the AHAB component is a complete subsystem on some NXP processors. They are responsible for verifying the digital signatures that are included as part of the product software. They also ensure that when the processor is configured as a secure device, no unauthenticated code is allowed to run.

On NXP processors that support the HAB/AHAB feature, you can also use encrypted boot to provide image cloning protection. Also, depending on the use case, image confidentiality can also be used. The secure components can be used to authenticate boot chain components, not only in the initial stage but also in later stages. The HAB/AHAB feature is agnostic to the bootloader and the operating system.

Figure 1 shows a generic boot chain example explaining the use of the HAB feature.

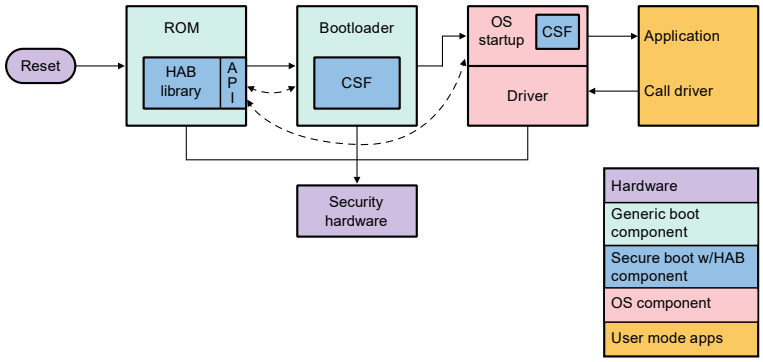

Figure 1. Generic boot flow using HAB

Figure 2 shows how to use the AHAB feature in a boot chain on the i.MX 8 and i.MX 8x family devices.

All information provided in this document is subject to legal disclaimers.

© 2024 NXP B.V. All rights reserved.

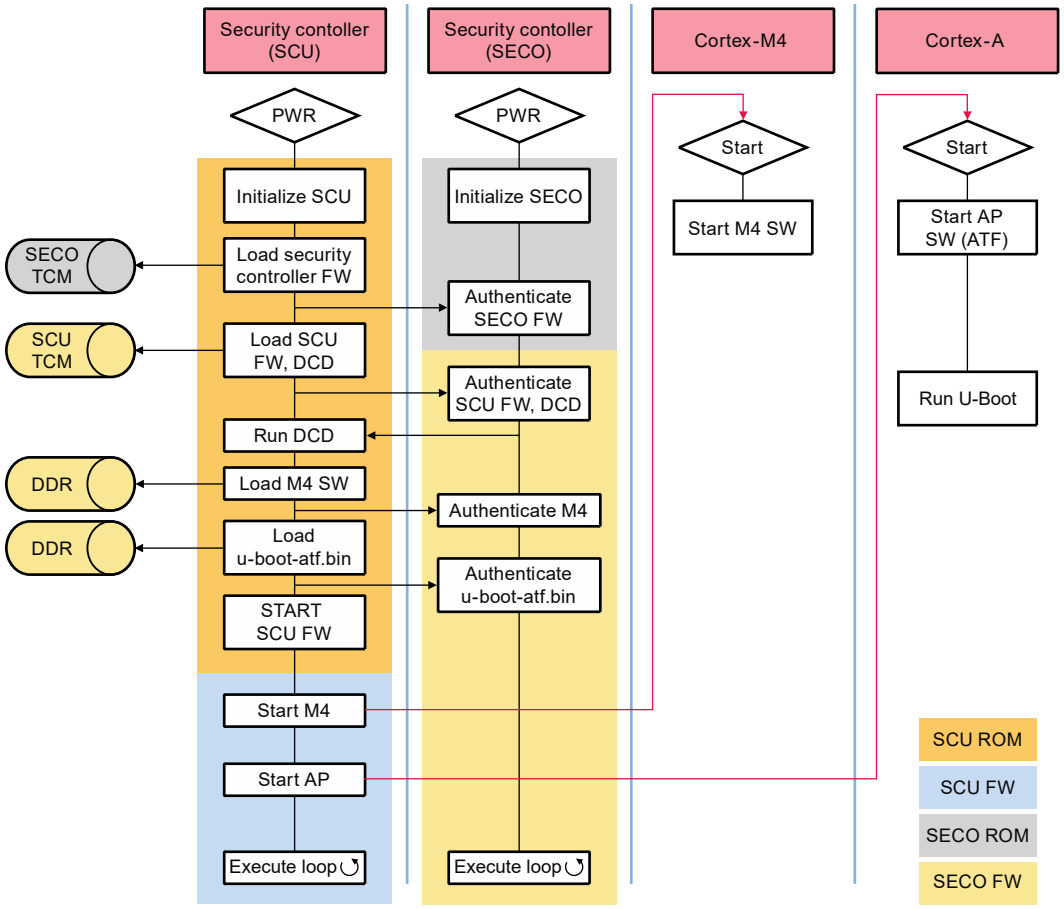

Figure 2. Boot flow using AHAB (SECO ROM and firmware)

As compared with the i.MX 8 and i.MX 8X devices, an i.MX 8XLite device has a new V2X subsystem to accelerate/offload V2X related cryptographic operations.

Figure 3 shows the i.MX 8DXL boot flow.

All information provided in this document is subject to legal disclaimers.

© 2024 NXP B.V. All rights reserved.

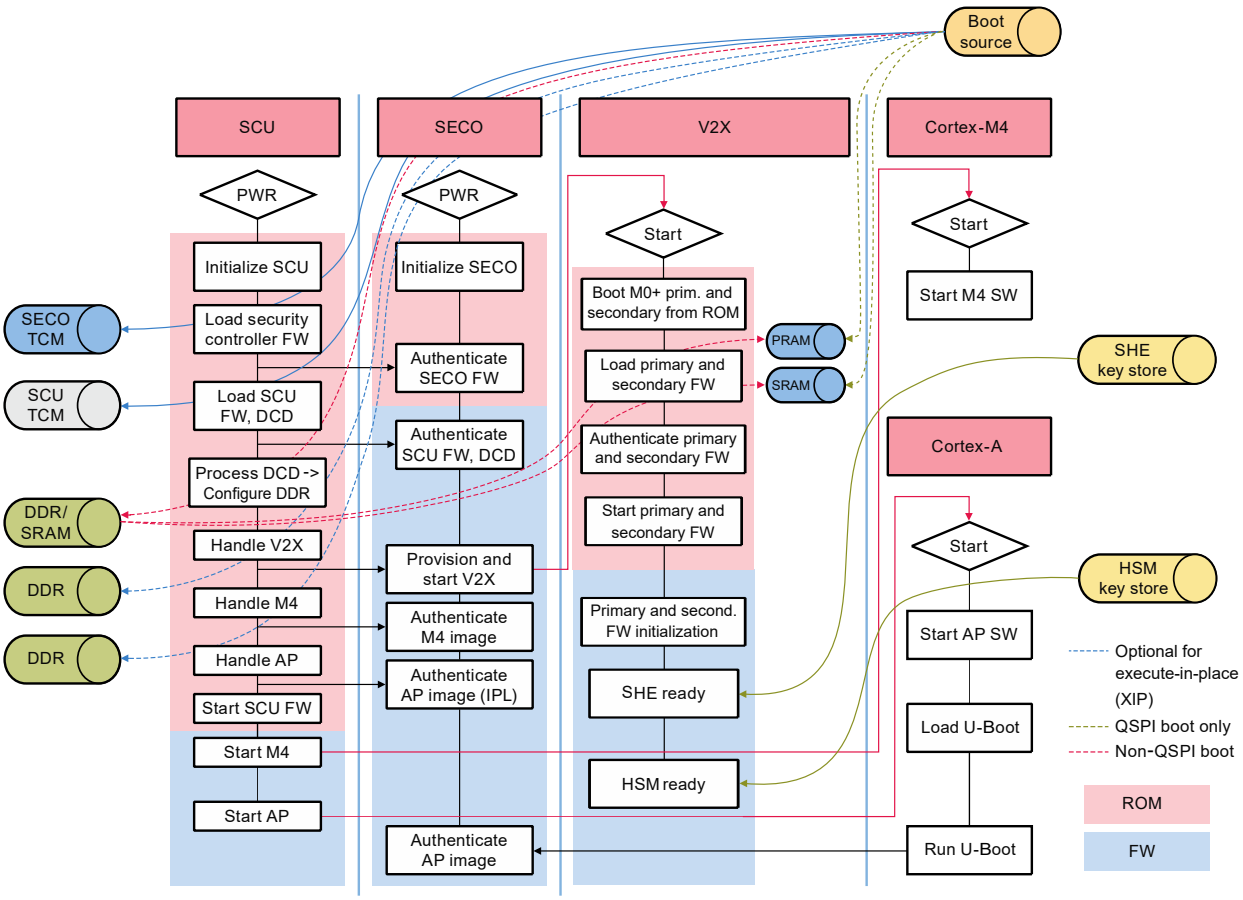

Figure 3. i.MX 8DXL boot flow

Figure 4 provides an example high-level boot flow diagram for i.MX 8ULP and i.MX 9x devices using dual boot mode. The devices also support other boot modes.

All information provided in this document is subject to legal disclaimers.

© 2024 NXP B.V. All rights reserved.

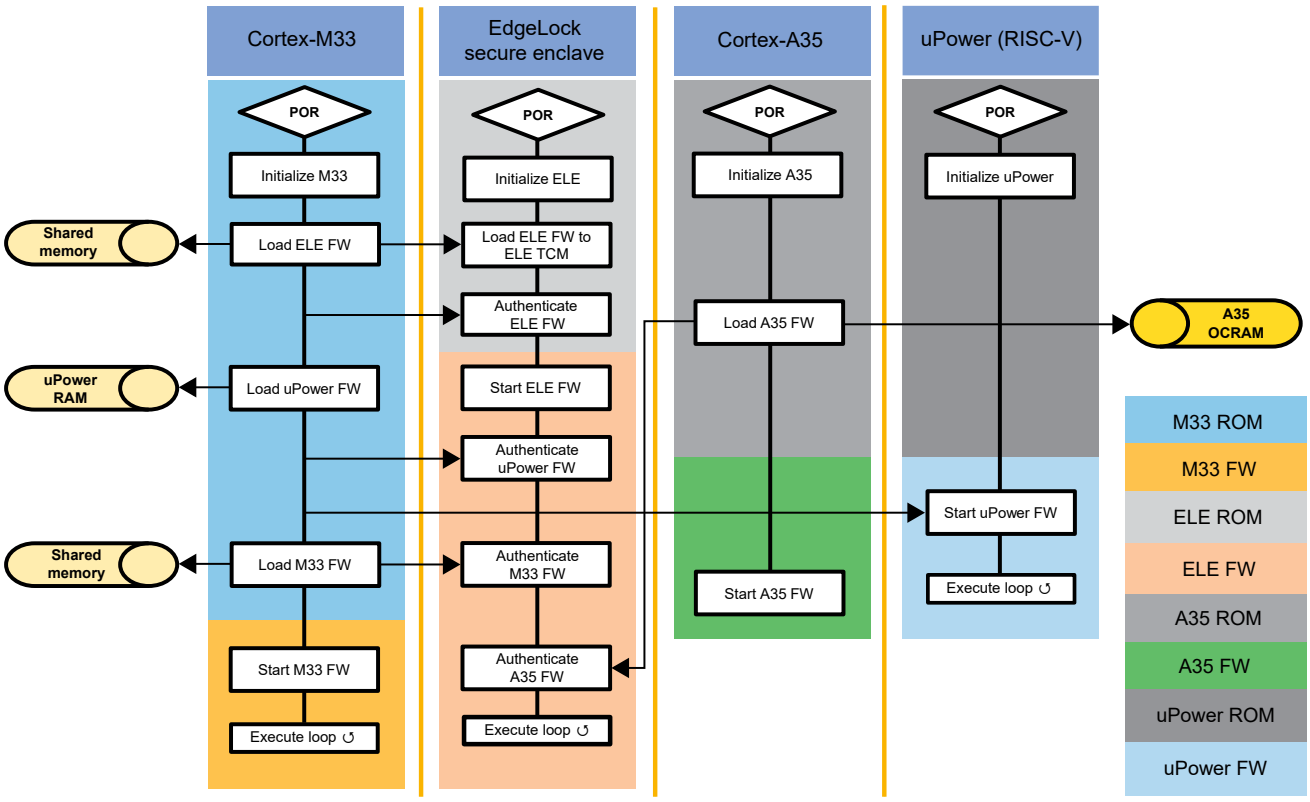

Figure 4. Dual boot mode

The i.MX 93 device supports many boot modes, including the following boot modes:

- Single boot
- Low-power boot

Figure 5 provides an example boot flow diagram for the i.MX 93 device using single boot mode. For details on the i.MX 93 low-power boot mode, refer to *i.MX 93 Applications Processor Reference Manual*.

All information provided in this document is subject to legal disclaimers.

© 2024 NXP B.V. All rights reserved.

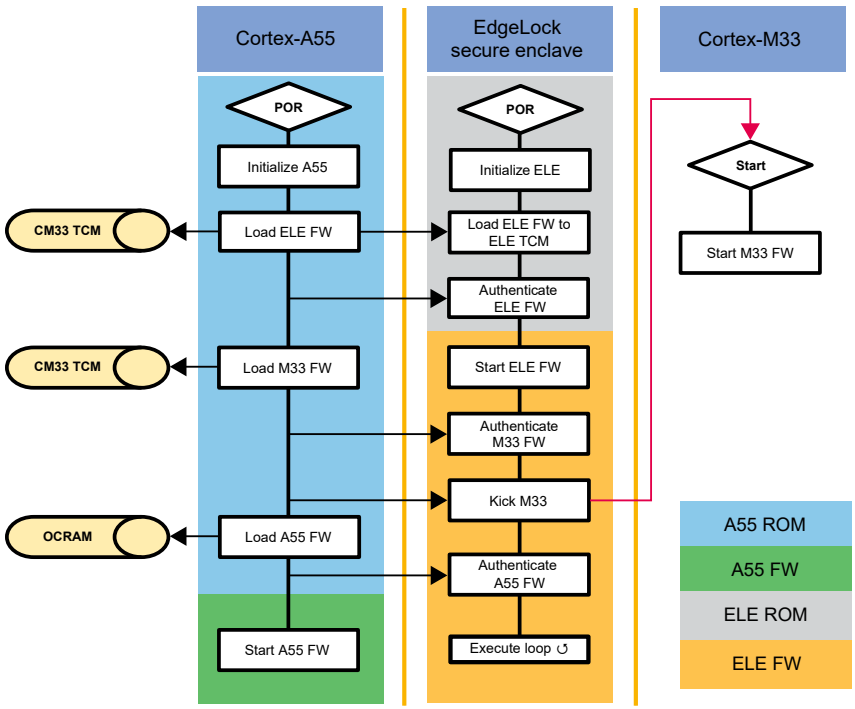

Figure 5. Single boot mode

Note: *EdgeLock secure enclave (ELE) firmware is optional in the i.MX 8ULP and i.MX 93 devices.*

The secure boot process starts with the ROM reading eFuses to determine the security configuration of the SoC and the type of the boot device. Then, the ROM loads the images to the memory.

For HAB, the bootloader image contains:

- Bootloader
- Commands that the HAB uses to verify Command Sequence File (CSF) data, including the image, digital signature data, and public key certificate data

The CSF data is generated offline using the code signing tool (CST), which is explained in Section 2.1.2.

For AHAB, the boot image contains:

- User-provided images (for user-programmable cores)
- A container header
- A signature block that the AHAB uses to verify the image, digital signatures, and public key certificate data

The container header is generated offline using the imx-mkimage tool (for details on how to use the imx-mkimage tool, see i.MX Linux User's Guide). The CST generates the signature block offline.

After the ROM has completed loading the images, execution is passed to the secure components that verify the signatures. If signature verification fails, execution is not allowed to leave the ROM for securely configured SoCs. The exact behavior after signature verification failure at the ROM stage depends on the SoC. If all signatures, including image decryption, are successful; the execution is passed to the next images. These images can perform similar steps to verify the next boot stage by calling back into the secure API.

The ROM, HAB, and AHAB cannot be changed, so they can be considered as trusted software components. Therefore, these components can be used to establish a secure boot chain.

HAB and AHAB require the use of physical addresses. Therefore, if one MMU and one level 2 cache are enabled in the bootloader stage, the address translation must be unchanged. It ensures that all boot

All information provided in this document is subject to legal disclaimers.

© 2024 NXP B.V. All rights reserved.

components provide HAB or AHAB with physical addresses. After all boot components have been verified, HAB and AHAB are not needed. Then, the MMU and level 2 cache can be reconfigured as per the requirement of the operating system (OS).

The ROM/HAB/AHAB library integration also provides access to the APIs that may be called for image verification by the boot components outside the ROM. The exact implementation of the APIs depends on the NXP processor in use. Refer to the processor reference manual for specific details.

The following two major versions of the secure components exist on NXP processors:

- HAB version 4 (HABv4): Supports the flow shown in Figure 1.
- AHAB version: Supports the flow shown in Figure 2.

HABv4 and AHAB use public key signature verification to ensure that product code is authentic. Table 1 highlights some differences between these two versions. To determine whether to use HAB or AHAB for an NXP processor, see the reference manual of the NXP processor.

Table 1. Differences between HABv4 and AHAB

Feature	HABv4	AHAB
Image authentication	Yes	Yes
Super root key	Multiple, revocable, and fused hash	Multiple, revocable, and fused hash
Public key type	• ECC-P256, ECC-P384, and ECC-P521 (only applicable to i.MX 8M Plus) • RSA-1024, RSA-2048, RSA-3072, and RSA-4096	• ECC-P256, ECC-P384, and ECC-P521 • RSA-2048, RSA-3072, and RSA-4096 AHAB version 2 additionally supports: • PQC: – Dilithium 3 – Dilithium 5 – ML-DSA-65 – ML-DSA-87 • Hybrid: – ECC-P384 + Dilithium 3 – ECC-P384 + ML-DSA-65 – ECC-P521 + Dilithium 5 – ECC-P521 + ML-DSA-87
Certificate format	X.509	NXP proprietary
Signature format	• CMS (PKCS#1) • CMS (ECDSA) (HAB 4.5 and later)	• PKCS#1 • ECDSA (raw format, no DER encoding) AHAB version 2 additionally supports: • Dilithium 3 (raw) • Dilithium 5 (raw) • ML-DSA-65 (raw) • ML-DSA-87 (raw) • Hybrid: ECDSA (raw format, no DER encoding) + PQC (raw)
Hash algorithm	SHA-256	• SHA-256 • SHA-384 • SHA-512 AHAB version 2 additionally supports: • SHA3-256

All information provided in this document is subject to legal disclaimers.

© 2024 NXP B.V. All rights reserved.

Table 1. Differences between HABv4 and AHAB*...continued*

Feature	HABv4	AHAB
		• SHA3-384 • SHA3-512 • SHAKE128 output 256 bits • SHAKE256 output 512 bits
Image decryption	Yes (HABv4.1 and later)	Yes (SECO firmware v2.3.0 and later, ELE 0.0.10 and later)
Image decryption algorithm	AES-CCM	AES-CBC
Image decryption key blob algorithm	NXP proprietary	NXP proprietary
Wrapped key format	CAAM blob — The secret keys are stored in the CAAM secure RAM partition.	• CAAM blob (for SECO-based devices) — The secret keys are stored in the CAAM secure RAM partition. • ELE blob (for ELE-based devices) — The secret keys are stored in the ELE enclave.
Secret key type	AES-128/192/256	AES-128/192/256
Decryption algorithm	AES-CCM — Authenticated decryption	AES-CCM — Authenticated decryption
Unlock commands	For example: • Field Return Fuse • Revocation Fuses • Secure JTAG	Not applicable
CSF commands	Yes	Only applicable on the tool (CST)

2.1.1.1 Secure component API

To continue the secure boot chain, the boot components outside the ROM must be able to call back into the HAB or AHAB. An application programming interface (API) is available for secure components. The API has two versions:

- One for HABv4. Information on the HABv4 API can be found in *High Assurance Boot Version 4 API Reference Manual*.
- The other for AHAB. Information on the AHAB API can be found in the i.MX 8QXP/QM SECO API or the i.MX 8ULP / i.MX 9 ELE API document.

2.1.2 CST

Performing cryptographic signatures involves the following participants:

- Certificate authority (CA), which is responsible for protecting the top-level CA key and certifying lower-level code signing keys
- Signature authority (SA), which is responsible for signing the code
- Manufacturer, who is responsible for requesting digital signatures across the product software

Figure 6 illustrates the roles of the participants involved in the code signing process.

All information provided in this document is subject to legal disclaimers.

© 2024 NXP B.V. All rights reserved.

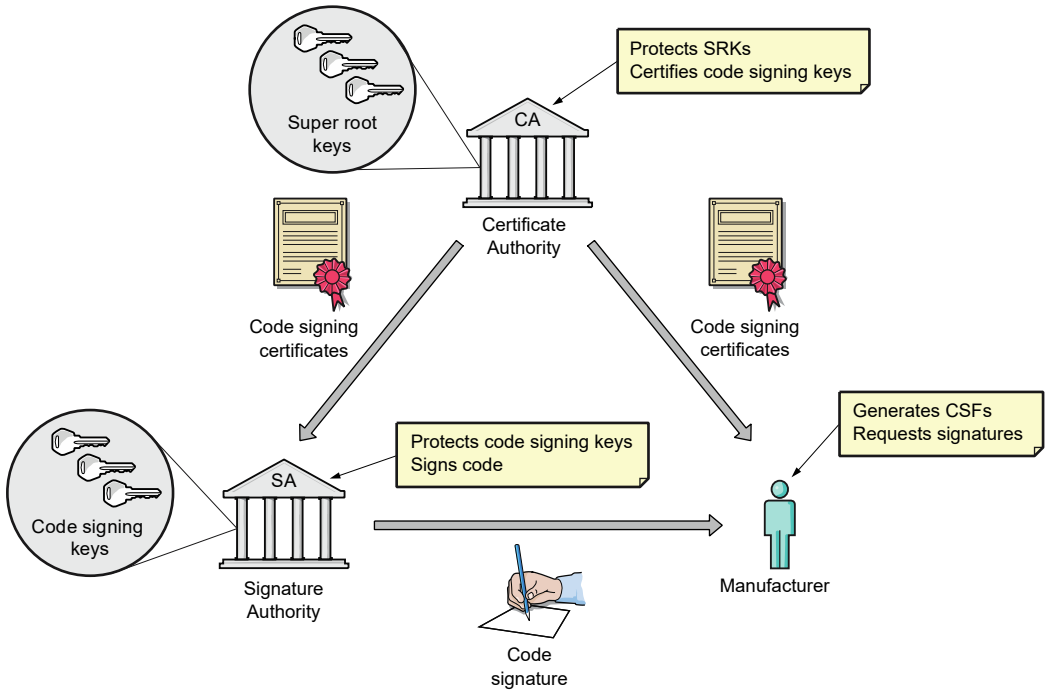

Figure 6. Generic code signing participants

The CST is a set of command-line tools residing on a host computer that serves as both the certificate authority (CA) and the signature authority (SA). It allows manufacturers to control all aspects of the signing process.

The CST can be used to perform the following functions:

- CA function (establish a public key infrastructure (PKI) tree of keys and certificates required for code signing)
- SA function (generate digital signatures across the data provided by the user)

The signatures generated by the CST can be included as part of the end-product software image. The secure components verify the signatures on the NXP processor at boot time.

Figure 7 shows how the CST is used to generate data, including signatures, certificates, and CSF commands (HAB only). The secure components in the ROM use this data to validate the product software.

The CST takes two main inputs:

- One or more binary images of the product software to be signed
- A Command Sequence File (CSF). The CSF description file contains instructions to be followed by the CST, for example:
  - Which areas of the binary image to sign
  - Which keys to use for signing the image

The CST takes these inputs and generates binary data, which includes signatures, certificates, and CSF commands (HAB only). The binary data can be attached to the product software to create a signed image. This document covers how to generate the keys, certificates, and CSF description files; and how to run the CST executable.

All information provided in this document is subject to legal disclaimers.

© 2024 NXP B.V. All rights reserved.

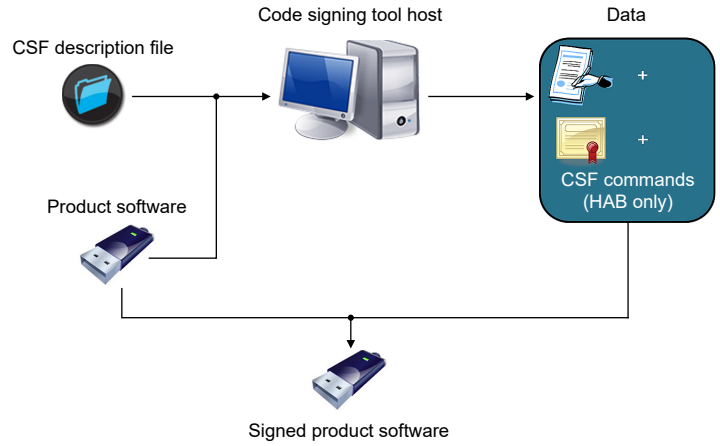

Figure 7. Code signing tool — Digital signatures

On some NXP processors that support HABv4 or AHAB, encrypted boot can also be used. Figure 8 shows the encrypted boot process with the CST.

The encrypted boot use case is similar to the signed image generation use case, except for the following two major differences:

- The binary image is both decrypted and authenticated using a symmetric key, rather than signed using a private asymmetric key.
- The CST generates a one-time Advanced Encryption Standard (AES) data encryption key (DEK), which is used to encrypt the image.

Note: *When performing an encrypted boot, digital signatures are still required. You can find an example CSF description file for encrypted boot in Section 6.3.3.*

The DEK is independent of the public keys used for code signing. The DEK output from the CST is protected but is not in the final form required for an encrypted boot on NXP processors.

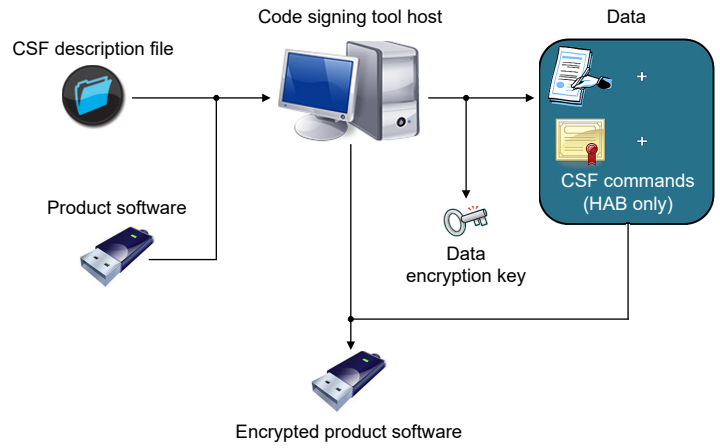

Figure 8. Code signing tool — Encrypted boot

During the OEM manufacturing stages of a processor, a cryptographic blob of the DEK must be created and attached to the image on the boot device. The DEK blob is created using the device unique key embedded in the NXP processor that is only readable by the on-chip encryption engine. Therefore, the DEK blob is unique for each IC, though the DEK is common to all ICs sharing the encrypted image.

Figure 9 provides an overview of DEK blob creation. The remaining details on DEK blob creation are outside the scope of the CST and this document.

All information provided in this document is subject to legal disclaimers.

© 2024 NXP B.V. All rights reserved.

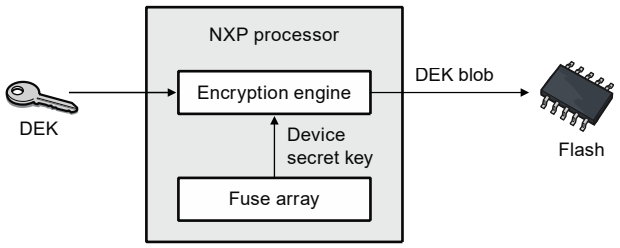

Figure 9. DEK blob creation

All information provided in this document is subject to legal disclaimers.

© 2024 NXP B.V. All rights reserved.

3 Installation

This section describes the installation of the CST code-signing client files.

3.1 CST package contents and installation

The CST is delivered in an archive file, which contains a version for Linux and a version for Windows. For macOS, the CST is no longer released with supported binaries, but the build system support for the macOS target is retained in the source code. The archive contains a `Software_Content_Register_CST.txt` file that lists the entire contents of the archive.

3.1.1 Linux system requirements for CST

Table 2 lists the software requirements for installing CST on a Linux host. If any component is missing, check with your system administrator.

Table 2. Software requirement checklist for Linux

Available?	Required component
❑	A Linux distribution. CST is supported on Ubuntu versions from 18.04 to 22.04. However, it may also work on other Linux distributions. You can determine your current Linux distribution by viewing the information shown on the Linux login screen.
❑	The recommended OpenSSL version for CST is 3.2.0. The previous OpenSSL version (1.1.1) reached its end-of-life (EOL) in September 2023. OpenSSL is required for the included scripts to generate public key infrastructure (PKI). You can determine your OpenSSL version by running the `openssl version` command. OpenSSL is available at http://www.openssl.org/.

Note: *Using the Linux OS, the NXP reference CST generates random numbers to be used as keys for encrypted boot. Therefore, the Linux host on which the reference CST is installed must have sufficient sources of entropy. Usually, it requires multiple entropy sources, such as keyboard input, mouse input, and network packet arrival time. Running the CST without these sources of entropy causes lengthy delays in seeding the Linux random number generator.*

3.1.2 Windows system requirements for CST

Table 3 lists the software requirements for installing CST on a Windows host. If any component is missing, check with your system administrator.

Table 3. Software requirement checklist for Windows

Available?	Required component
❑	CST is supported on Microsoft Windows 7 (32-bit) and Windows 10 (64-bit). You can determine the version of the Windows operating system installed on your Windows computer by checking the properties of the computer.
❑	The recommended OpenSSL version for CST is 3.2.0. The previous OpenSSL version (1.1.1) reached its end-of-life (EOL) in September 2023. OpenSSL is required for the included scripts to generate public key infrastructure (PKI). You can determine your OpenSSL version by running the `openssl version` command. OpenSSL is available at http://www.openssl.org/. ***Note:*** *Sometimes, the OpenSSL Windows installer does not set the `PATH` environment variable automatically. Ensure that this variable is set to the OpenSSL bin directory.*

All information provided in this document is subject to legal disclaimers.

© 2024 NXP B.V. All rights reserved.

Note: *Using the Windows OS, the NXP reference CST generates random numbers to be used as keys for encrypted boot. Therefore, the Windows host on which the reference CST is installed must have sufficient sources of entropy. Usually, it requires multiple entropy sources, such as keyboard input, mouse input, and network packet arrival time. Running the CST without these sources of entropy causes lengthy delays in seeding the Windows random number generator.*

3.1.3 Unpacking files

You can unpack the CST archive to any desired location on your host computer. The following is an example for Linux and it assumes that the client archive was saved in a directory named `/home/<username>/cst`:

```
$ cd /home/<username>/cst/
$ tar -zxvf <release package name>.tgz
```

For a native Windows installation without Cygwin:

1. Save the `<CST release package name>.zip` file in a folder `c:/cst`.
2. Use WinZip or an equivalent archiver program to extract the CST client from `<CST release package name>.zip`. Extracting the ZIP file creates the following directories:
   - `ca/`: Contains the OpenSSL configuration files. These configuration files are used when generating signing keys and certificates with the OpenSSL command-line tool.
   - `src/`: Contains the CST source code.
   - `docs/`: Contains *Code Signing Tool User Guide* and *High Assurance Boot Version 4 API Reference Manual*.
   - `examples/`: Includes examples of CSF files for HABv4 and AHAB, as well as signed messages for AHAB.
   - `crts/`: Contains the public key certificates used for signing. Initially, this directory is empty.
   - `keys/`: Contains the private key files used for signing. Initially, this directory contains the following scripts for generating the PKI tree:
     - `hab4_pki_tree.sh`: Generates a series of keys and certificates on a Linux or macOS machine for use with an NXP processor that supports HABv4.
     - `hab4_pki_tree.bat`: Generates a series of keys and certificates on a Windows machine for use with an NXP processor that supports HABv4.
     - `ahab_pki_tree.sh`: Generates a series of keys and certificates on a Linux or macOS machine for use with an NXP processor that supports AHAB.
     - `ahab_pki_tree.bat`: Generates a series of keys and certificates on a Windows machine for use with an NXP processor that supports AHAB.
     - `add_key.sh`: Adds new keys to an existing HABv4 or AHAB PKI tree.
     - `hsm_hab4_pki_tree.sh`: Generates a series of keys and certificates on a PKCS#11-enabled HSM token for use with an NXP processor that supports HABv4.
     - `hsm_ahab_pki_tree.sh`: Generates a series of keys and certificates on a PKCS#11-enabled HSM token for use with an NXP processor that supports AHAB.
     - `hsm_add_key.sh`: Adds new keys to an existing HABv4 or AHAB PKI tree within a PKCS#11-enabled token.
   - `linux32/`: Contains the CST executables for a 32-bit Linux OS:
     - `bin/cst`: Represents the CST executable to be used for signing the code.
     - `bin/srktool`: Generates the SRK table and eFuse files for HABv4 or AHAB.
     - `bin/hab_log_parser`: Parses the HAB-persistent memory dumps and prints the HAB events.
     - `bin/hab_csf_parser`: Parses a CSF binary from either a signed image or a standalone CSF binary to output debug data, including the extraction of certificates, signatures, and the SRK table.

All information provided in this document is subject to legal disclaimers.

© 2024 NXP B.V. All rights reserved.

  - bin/hab4_pki_tree: Generates a series of keys and certificates for use with an NXP processor that supports HABv4.
  - bin/ahab_pki_tree: Generates a series of keys and certificates with post-quantum cryptography (PQC) support for use with an NXP processor that supports AHAB.
  - bin/hab4_mac_dump: A tool for dumping the MAC data location and size from a HABv4 CSF binary data file.
  - bin/ahab_split_container: Splits NXP-signed AHAB container into ELE and V2X containers, enabling the OEM double signing feature.
  - bin/ahab_signed_message: Generates signed messages for AHAB devices.
  - bin/hab4_image_verifier: Parses and verifies HABv4 images for debugging purposes.
  - bin/ahab_image_verifier: Parses and verifies AHAB containers for debugging purposes.
- linux64/: Contains the CST executables for a 64-bit Linux OS:
  - bin/cst: Represents the CST executable to be used for signing the code.
  - bin/srktool: Generates the SRK table and eFuse files for HABv4 or AHAB.
  - bin/hab_log_parser: Parses the HAB-persistent memory dumps and prints the HAB events.
  - bin/hab_csf_parser: Parses a CSF binary from either a signed image or a standalone CSF binary to output debug data, including the extraction of certificates, signatures, and the SRK table.
  - bin/hab4_pki_tree: Generates a series of keys and certificates for use with an NXP processor that supports HABv4.
  - bin/ahab_pki_tree: Generates a series of keys and certificates with PQC support for use with an NXP processor that supports AHAB.
  - bin/hab4_mac_dump: A tool for dumping the MAC data location and size from a HABv4 CSF binary data file.
  - bin/ahab_split_container: Splits NXP-signed AHAB container into ELE and V2X containers, enabling the OEM double signing feature.
  - bin/ahab_signed_message: Generates signed messages for AHAB devices.
  - bin/hab4_image_verifier: Parses and verifies HABv4 images for debugging purposes.
  - bin/ahab_image_verifier: Parses and verifies AHAB containers for debugging purposes.
- mingw32/: Contains the 32-bit CST executables for MS Windows:
  - bin/cst.exe: Represents the CST executable to be used for signing the code.
  - bin/srktool.exe: Generates the SRK table and eFuse files for HABv4 or AHAB.
  - bin/hab_log_parser.exe: Parses the HAB-persistent memory dumps and prints the HAB events.
  - bin/hab_csf_parser.exe: Parses a CSF binary from either a signed image or a standalone CSF binary to output debug data, including the extraction of certificates, signatures, and the SRK table.
  - bin/hab4_pki_tree: Generates a series of keys and certificates for use with an NXP processor that supports HABv4.
  - bin/ahab_pki_tree.exe: Generates a series of keys and certificates with PQC support for use with an NXP processor that supports AHAB.
  - bin/hab4_mac_dump.exe: A tool for dumping the MAC data location and size from a HABv4 CSF binary data file.
  - bin/ahab_split_container: Splits NXP-signed AHAB container into ELE and V2X containers, enabling the OEM double signing feature.
  - bin/ahab_signed_message.exe: Generates signed messages for AHAB devices.
  - bin/hab4_image_verifier.exe: Parses and verifies HABv4 images for debugging purposes.
  - bin/ahab_image_verifier.exe: Parses and verifies AHAB containers for debugging purposes.
- Mingw64/: Contains the 64-bit CST executables for MS Windows:
  - bin/cst.exe: Represents the CST executable to be used for signing the code.
  - bin/srktool.exe: Generates the SRK table and eFuse files for HABv4 or AHAB.
  - bin/hab_log_parser.exe: Parses the HAB-persistent memory dumps and prints the HAB events.

All information provided in this document is subject to legal disclaimers.

© 2024 NXP B.V. All rights reserved.

- `bin/hab_csf_parser.exe`: Parses a CSF binary from either a signed image or a standalone CSF binary to output debug data, including the extraction of certificates, signatures, and the SRK table.
- `bin/hab4_pki_tree`: Generates a series of keys and certificates for use with an NXP processor that supports HABv4.
- `bin/ahab_pki_tree.exe`: Generates a series of keys and certificates with PQC support for use with an NXP processor that supports AHAB.
- `bin/hab4_mac_dump.exe`: A tool for dumping the MAC data location and size from a HABv4 CSF binary data file.
- `bin/ahab_split_container`: Splits NXP-signed AHAB container into ELE and V2X containers, enabling the OEM double signing feature.
- `bin/ahab_signed_message.exe`: Generates signed messages for AHAB devices.
- `bin/hab4_image_verifier.exe`: Parses and verifies HABv4 images for debugging purposes.
- `bin/ahab_image_verifier.exe`: Parses and verifies AHAB containers for debugging purposes.

After unpacking the archive, you can start using the CST.

All information provided in this document is subject to legal disclaimers.

© 2024 NXP B.V. All rights reserved.

4 Key and certificate generation

After the CST installation is complete, the first step in signing the code is to generate the private keys and certificates. CST is not delivered with keys or certificates because keys and certificates are different for each manufacturer / product line.

The NXP reference CST can be used to generate keys using:

- Either an OpenSSL command-line tool and a set of shell scripts for Linux / a set of batch files for Windows
- Or a set of binaries prelinked with the OpenSSL library

Note: *Due to the restrictions of OpenSSL versions (deprecation of OpenSSL version 1) and the increasing complexity of dependencies (for PQC support), the scripts may be removed in future CST releases. It will shift reliance solely to the tools provided within the package, eliminating the need to use the OpenSSL application in CLI mode.*

In this process, OpenSSL acts as the CA component shown in Figure 6. The scripts/tools contain the code to generate a PKI tree of keys and certificates. Because the HABv4 and AHAB each produces a different PKI tree structure; therefore, each of them requires a separate set of scripts/tools to generate an initial PKI tree.

Moreover, the PKI tree structure is also different for the final public key certificate format. HABv4 and AHAB require X.509 format certificates (see Table 24 for more details).

The key and certificate generation scripts/tools provided with CST are for reference purposes. You can use them as a reference to develop your own scripts/tools for generating keys and certificates with OpenSSL.

For post-quantum cryptography (PQC) and hybrid key generation, you may need to recompile and install the Open Quantum Safe (OQS) provider for OpenSSL along with its dependent libraries on your host computer. CST provides binaries that mimic traditional PKI key generation scripts but with PQC support, as these binaries are already prebuilt with the necessary providers and libraries.

The following are the file naming conventions for keys and certificates:

- For keys: `<keyname>_key.<ext>`
- For certificates: `<keyname>_crt.<ext>`

Note: *The NXP reference CST requires a one-to-one mapping between the keys in the `/keys` directory and the certificates in the `/crts` directory. For example, a key named `SRK1_sha256_2048_65537_v3_ca_key.der` in the `/keys` directory must have a corresponding certificate `SRK1_sha256_2048_65537_v3_ca_crt.der` in the `/crts` directory.*

4.1 Generating HABv4 keys and certificates

This section covers the key and certificate generation process for HABv4.

Note: *When using the encrypted boot feature, digital signatures are still required.*

The data structures used by ROM and HAB cannot be encrypted but they must be signed using a valid digital signature. In addition, the CST generates a new symmetric key dynamically for each Install Secret Key / Decrypt Data command pair. These symmetric keys are an output of the CST and they are encrypted using a supplied public key. You can find an example CSF file for encrypted boot in Section 6.3.3.

4.1.1 HABv4 PKI tree

To generate the tree structure for HABv4, use one of the following scripts (based on the host type):

- The `hab4_pki_tree.sh` script file for Linux
- The `hab4_pki_tree.bat` batch file for Windows

All information provided in this document is subject to legal disclaimers.

© 2024 NXP B.V. All rights reserved.

The script generates a HABv4 PKI tree in the /keys directory of the NXP reference CST, as shown in Figure 10.

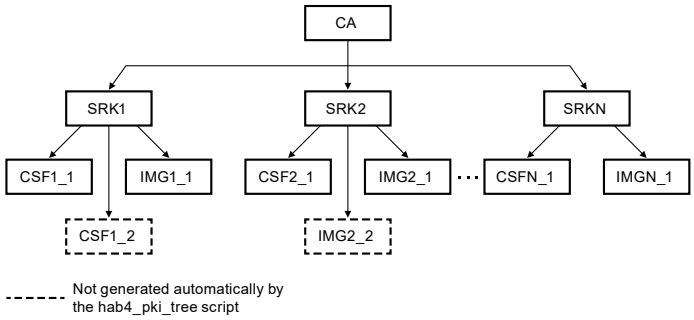

Figure 10. HABv4 PKI tree

A HABv4 PKI tree consists of the following keys and certificates:

- CA key: It is the top most key and is only used for signing SRK certificates.
- SRK: It is the root key for HAB code signing keys. The cryptographic hash of a table of SRKs is burned to a set of one-time programmable eFuses to establish a root of trust. Only one of the SRKs can be selected from the table for use with the NXP processor per reset cycle. The SRK selection is done based on a parameter in the Install SRK CSF command (see Section 6.2.2). The SRK can be used only for signing the certificate data of the subordinate keys.
- CSF: It is a subordinate key of the SRK that is used to verify the signature of the CSF commands.
- IMG: It is a subordinate key of the SRK key that is used to verify the signatures across the product software.
  Note: *The CSF and IMG keys are not generated for a fast authentication PKI tree.*

The hab4_pki_tree.sh script generates a basic tree with up to four SRKs. For each SRK, one CSF key and one IMG key are also generated. Using a separate script, more keys can be added to the tree later. Moreover, you can replace the OpenSSL and the hab4_pki_tree.sh script with an alternative key generation solution. How to create such a solution is outside the scope of this document.

If you use an alternative key generation scheme, the scheme must follow these constraints:

- Keys must be in the PKCS#8 format.
- Certificates must be in the X.509 format, and they must follow the certificate profile specified by HABv4. Keys and certificates must follow the file naming convention specified in Section 4.

4.1.2 Running hab4_pki_tree script example

The hab4_pki_tree script can be run in one of the following modes:

- Interactive
- Command-line interface (CLI)

The following are the common steps to generate a HABv4 PKI tree in Interactive or CLI mode for Linux. The steps for Windows are similar, with a major difference that in Windows, hab4_pki_tree.bat is used in place of hab4_pki_tree.sh:

1. Run the cd command:

```
cd <CST Installation Path>/keys
```

2. Using any text editor, create a file with the name serial in the /keys directory and add a number (for example, 12345678) to it. OpenSSL uses this number as the initial certificate serial number.

All information provided in this document is subject to legal disclaimers.

© 2024 NXP B.V. All rights reserved.

3. Using any text editor, create a file with the name `key_pass.txt` in the `/keys` directory. Use this file to store your pass phrase to protect the HAB code signing private keys. Write the pass phrase twice (in the first and second lines) in the file, for example:
   my_pass_phrase
   my_pass_phrase
   Note: *Before running the `hab4_pki_tree.sh` script, generate the `serial` and `key_pass.txt` files to avoid facing issues in generating the PKI tree.*
   CAUTION: *Ensure that the pass phrase for the private keys is protected because impacted keys cannot be used to sign the code.*
   Note: *OpenSSL enforces a minimum character length of 4 for a pass phrase.*
4. Before running the `hab4_pki_tree.sh` script, include OpenSSL in your search path:

```
> openssl version
```

4.1.2.1 Running `hab4_pki_tree` script in Interactive mode

Running the `hab4_pki_tree.sh` script (for Linux) or the `hab4_pki_tree.bat` script (for Windows) in Interactive mode prompts you with the following questions:

- Do you want to use an existing CA key (y/n)?
  - Choose 'n' unless you already have an existing CA key.
  - If you choose 'y', the script asks you to provide the filenames (including file paths) of the CA key and corresponding CA public key certificate.
- Select the key type (possible values: rsa, rsa-pss, ecc)?
  - It indicates the type of the keys in the tree.
  - If you choose "ecc", the script asks you to provide the elliptic curve (EC) to be used. For HABv4, the supported elliptic curves are P-256, P-384, and P-521.
  - If you choose "rsa" or "rsa-pss", the script asks you to enter the length in bits for the RSA keys in the tree. For HABv4, the supported RSA key lengths are 1024 bits, 2048 bits, 3072 bits, and 4096 bits.
  - All keys in the tree are generated with the same length.
- Enter the PKI tree duration (in years):
  - It defines the validity period of the corresponding certificates.
- How many super root keys (SRKs) should be generated?
  - The script can generate up to four SRKs. It allows you to include up to four SRKs in a HABv4 SRK table. For more details, see Section 5.2.
- Do you want the SRK certificates to have the CA flag set?
  - Answer ‘y’ for a standard tree or ‘n’ for a fast authentication tree.

Code listing "Example usage of `hab4_pki_tree.sh` script" illustrates the use of the `hab4_pki_tree.sh` script.

Code listing: Example usage of `hab4_pki_tree.sh` script

```
$ ./hab4_pki_tree.sh
...
<snip>
...
Do you want to use an existing CA key (y/n)?: n

Key type options (confirm targeted device supports desired key type):
Select the key type (possible values: rsa, rsa-pss, ecc)?: rsa
Enter key length in bits for PKI tree: 2048
Enter PKI tree duration (years): 10
How many Super Root Keys should be generated? 1
```


All information provided in this document is subject to legal disclaimers.

© 2024 NXP B.V. All rights reserved.

```
Do you want the SRK certificates to have the CA flag set? (y/n)?: n
A default 'serial' file was created!
A default file 'key_pass.txt' was created with password = test!

++++++++++++++++++++++++++++++++++
+ Generating CA key and certificate +
++++++++++++++++++++++++++++++++++

Generating a RSA private key
..................................+++++
.....+++++
writing new private key to 'temp_ca.pem'
-----

++++++++++++++++++++++++++++++++++++++
+ Generating SRK key and certificate 1 +
++++++++++++++++++++++++++++++++++++++

.....
+++++
.....+++++
Using configuration from ../ca/openssl.cnf
Check that the request matches the signature
Signature ok
The Subject's Distinguished Name is as follows
commonName            :ASN.1 12:'SRK1_sha256_2048_65537_v3_usr'
Certificate is to be certified until Dec 12 21:44:07 2033 GMT (3650 days)

Write out database with 1 new entries
Data Base Updated
```

4.1.2.2 Running `hab4_pki_tree` script in CLI mode

Running the `hab4_pki_tree.sh` script (for Linux) or the `hab4_pki_tree.bat` script (for Windows) in Command-line interface (CLI) mode can be helpful when automation is needed. You can run the `hab4_pki_tree` script in CLI mode as follows:

```
./hab4_pki_tree.sh -existing-ca <y/n> [-ca-key <CA key name> -ca-cert <CA cert
 name>]
-use-ecc <y/n> -kl <ECC/RSA Key Length> -duration <years> -num-srk <1-4> -srk-ca
 <y/n>
```

where:

- `-existing-ca`: Choose whether to use an existing CA key. Valid inputs are:
  - y: Provide `-ca-key` with CA key filename and `-ca-cert` with CA public key certificate filename (including path information).
  - n: The existing CA key is not selected.
- `-use-ecc`: Choose whether to use Elliptic Curve Cryptography (ECC) or RSA. Valid inputs are:
  - y: ECC keys are generated for the PKI tree.
  - n: RSA keys are generated for the PKI tree.
- `-kl`: Enter the key length for the key type selected:
  - If `-use-ecc` is 'y', provide `-kl` with the elliptic curve length for the PKI tree. Possible values are p256, p384, and p521.

All information provided in this document is subject to legal disclaimers.

© 2024 NXP B.V. All rights reserved.

- If `-use-ecc` is 'n', provide `-kl` with the RSA length in bits for the PKI tree. For HABv4, possible values (in bits) are 1024, 2048, 3072, and 4096.
- `-duration`: Enter PKI tree duration (in years).
- `-num-srk`: Enter up to four SRKs (1 – 4).
- `-srk-ca`: Specify whether you want the SRK certificate to have the CA flag set. Valid inputs are:
  - y: A standard PKI tree is created.
  - n: A fast authentication PKI tree is created.

For more information on these options, see Section 4.1.2.1.

At this point, the script generates the SRK, CSF, and IMG keys and certificates in the `/keys` and `/crts` directories, respectively. The generated keys exist in the PKCS#8 format (see Table 24 for more details) in both PEM and DER forms. The generated certificates are in the X.509 format (see Table 24 for more details) in both PEM and DER forms. The CST tool accepts key and certificate files in either PEM or DER form.

After the completion of the OpenSSL key and certificate generation process, several files are left over, including `12345678.pem`, `serial.old`, and `index.txt.attr`.

At this point, you should have all the information related to the keys and certificates for signing an image using HABv4.

4.1.3 Generating HABv4 SRK tables and eFuse hash

The previous section (Section 4.1.2) explained the steps to generate the keys and certificates for a HABv4 PKI tree. The next step is to generate a HABv4 SRK table and corresponding hash value for burning to eFuses on the SoC.

In HABv4, up to four SRKs can be included in a signed image, though only one can be used per reset cycle. The SRKs are kept in a table, and one of the SRKs is selected during the boot process. The Install SRK CSF command (see Section 6.2.2) selects one SRK from the table that is used to establish the root of trust.

Any of the SRKs can be selected from the table, without changing the `SRK_HASH` values burned into eFuses on the SoC. It is useful on NXP processors where multiple eFuses are available for SRK revocation. If one or more SRKs from an SRK table are compromised, you can burn eFuses corresponding to the compromised SRKs, making them unusable. The HAB library ensures that these SRKs are not used again. The next SRK in the table can be used to sign new images.

A minimum of one and a maximum of four SRKs can be placed in an SRK table. Only the first three SRKs can be revoked in a table. Therefore, you are recommended to use an SRK table with four keys so that you can have one SRK that cannot be revoked.

SRK tables are generated using the `srktool`. Using the four SRKs created in Section 4.1.2.2, an SRK table can be generated from the `/crts` directory, as shown in Code listing "SRK table and eFuse generation example".

Code listing: SRK table and eFuse generation example

```
$ ../linux64/bin/srktool -h 4 -t SRK_1_2_3_4_table.bin -e
 SRK_1_2_3_4_fuse.bin -d sha256 -c ./SRK1_sha256_2048_65537_v3_ca_crt.pem,./
SRK2_sha256_2048_65537_v3_ca_crt.pem,./SRK3_sha256_2048_65537_v3_ca_crt.pem,./
SRK4_sha256_2048_65537_v3_ca_crt.pem -f 1
Number of certificates    = 4
SRK table binary filename = SRK_1_2_3_4_table.bin
SRK Fuse binary filename  = SRK_1_2_3_4_fuse.bin
SRK Fuse binary dump:
SRK HASH[0] = 0x5B31CBE9
SRK HASH[1] = 0x6DE304C8
SRK HASH[2] = 0x99F821DE
```


All information provided in this document is subject to legal disclaimers.

© 2024 NXP B.V. All rights reserved.

```
SRK HASH[3] = 0x2803B237
SRK HASH[4] = 0xC8EF0FF8
SRK HASH[5] = 0x12F30689
SRK HASH[6] = 0xF38CE4A3
SRK HASH[7] = 0x39669C00
```

In this example:

- All four SRKs are included in the table.
- The SHA-256 hash value is generated with 32 bits of eFuse data per word. Some NXP processors require the hash value to be generated with 8 bits of eFuse data per word. In that case, use the `-f 0` option.
- The hash result in the resulting `SRK_1_2_3_4_fuse.bin` file is in little-endian format. It means that:
  - The first byte in the file corresponds to `SRK_HASH[255:248]` in the fuse map.
  - The last byte in the file corresponds to `SRK_HASH[7:0]` in the fuse map.

  Similarly, when using the `-f 0` option:
  - The first non-zero byte in the file corresponds to `SRK_HASH[255:248]` in the fuse map.
  - The last non-zero byte corresponds to `SRK_HASH[7:0]` in the fuse map.

Note: *Do not enter a whitespace between the ',' when specifying the SRKs in the `-c` or `--certs` option. Adding a whitespace causes all certificates specified after the first whitespace not to be included in the table and the resulting eFuse hash.*

4.1.4 Programming SRK hash value to eFuses

The previous section provided details on how to generate the SRK tables and the corresponding eFuse data. This section covers how to program the SRK hash value. The `hab4_pki_tree.sh` script computes the hash value using `srktool` and stores it in an eFuse file. The hash value stored in the eFuse file is burned to the `SRK_HASH` eFuse field on the SoC supporting HABv4.

The `SRK_1_2_3_4_fuse.bin` file from Code listing "SRK table and eFuse generation example" in Section 4.1.3 has the following contents:

```
e9cb315bc804e36dde21f89937b20328f80fefc88906f312a3e48cf3009c
```

This hash value must be burned into the `SRK_HASH` eFuse field of the SoC for HABv4, in the following order:

```
SRK HASH[0] = 0x5B31CBE9
SRK HASH[1] = 0x6DE304C8
SRK HASH[2] = 0x99F821DE
SRK HASH[3] = 0x2803B237
SRK HASH[4] = 0xC8EF0FF8
SRK HASH[5] = 0x12F30689
SRK HASH[6] = 0xF38CE4A3
SRK HASH[7] = 0x39669C00
```

To know the location of the `SRK_HASH` field, refer to the fuse map of the NXP processor in use.

4.1.5 Adding a key to a HABv4 PKI tree

A new key can be added to an existing HABv4 PKI tree by running one of the following scripts (based on the host type):

- The `add_key.sh` script file for Linux
- The `add_key.bat` batch file for Windows

The `add_key.sh`/`add_key.bat` script can be run in one of the following modes:

All information provided in this document is subject to legal disclaimers.

© 2024 NXP B.V. All rights reserved.

- Interactive
- Command-line interface (CLI)

4.1.5.1 Running `add_key` script in Interactive mode

Running the `add_key.sh` script (for Linux) or the `add_key.bat` script (for Windows) in Interactive mode prompts you with the following questions:

- Which version of HAB/AHAB do you want to generate the key for (4 = HABv4 / a = AHAB)?
- Enter new key name (for example, SRK5):
  - Specify a name for the new key, for example, SRK2 or CSF1_2.
- Enter new key type (ecc / rsa / rsa-pss):
  - Specify the type of the new key: ECC, RSA, or RSA-PSS
- Enter new key length in bits:
  - Specify the length of the new key in bits that matches the length of the signing key.
- Enter the certificate duration (in years):
  - Specify the validity period for the certificate to be generated.
- Is it an SRK key?
  - If you are generating a new SRK key, enter ‘y’; otherwise, enter ‘n’.
  - If you enter ‘n’, it implies that you are generating a CSF/IMG/SGK key.
- Enter `<key type>` signing key name:
  - If you are generating a new SRK key, `<key type>` is CA. Enter the path and filename of the CA key in the `/keys` directory that you want to use for generating the SRK key.
  - If you are generating a new CSF/IMG/SGK key, `<key type>` is SRK. Enter the path and filename of the SRK in the `/keys` directory that you want to use for generating the CSF/IMG/SGK key.
- Enter `<cert type>` signing certificate name:
  - If you are generating a new SRK certificate, the `<cert type>` is CA. Enter the path and filename of the CA certificate in the `/crts` directory that you want to use for generating the SRK certificate.
  - If you are generating a new CSF/IMG/SGK certificate, `<cert type>` is SRK. Enter the path and filename of the SRK certificate in the `/certs` directory that you want to use for generating the CSF/IMG/SGK certificate.

Using the keys generated in Section 4.1.2, a new SRK key can be added to the PKI tree, as shown in Code listing "Adding a new SRK to a HABv4 PKI tree example".

Code listing: Adding a new SRK to a HABv4 PKI tree example

```
$ ./add_key.sh
Which version of HAB/AHAB do you want to generate the key for (4 = HAB4 / a =
 AHAB)?: 4
Enter new key name (e.g. SRK5):
Enter new key type (ecc / rsa / rsa-pss): rsa
Enter new key length in bits: 2048
Enter certificate duration (years): 10
Is this an SRK key?: y
Do you want the SRK to have the CA flag set (y/n)?: y
Enter CA signing key name: CA1_sha256_2048_65537_v3_ca_key.pem
Enter CA signing certificate name: ../crts/CA1_sha256_2048_65537_v3_ca_crt.pem
.....................................................+++++
.................................+++++
Using configuration from ../ca/openssl.cnf
Check that the request matches the signature
Signature ok
The Subject's Distinguished Name is as follows
```


All information provided in this document is subject to legal disclaimers.

© 2024 NXP B.V. All rights reserved.

```
commonName            :ASN.1 12:'_sha256_2048_65537_v3_ca'
Certificate is to be certified until Dec 12 22:06:16 2033 GMT (3650 days)

Write out database with 1 new entries
Data Base Updated
```

4.1.5.2 Running `add_key` script in CLI mode

Running the `add_key.sh` script (for Linux) or the `add_key.bat` script (for Windows) in CLI mode can be helpful when automation is needed. You can run the `add_key` script in CLI mode as follows:

```
./add_key.sh -ver <4/a> -key-name <new key name> -kt <ecc/rsa> -kl <key length>
[-md <message digest>] -duration <years> -srk <y/n> [-srk-ca <y/n>]
-signing-key <CA/SRK signing key> -signing-crt <CA/SRK signing cert>
```

where:

- `-ver`: Enter the version of HAB/AHAB you want to generate the key for (4/a). Valid inputs are:
  - 4: HABv4 is selected.
  - a: AHAB is selected.
- `-key-name`: Enter the new key name.
- `-kt`: Enter the key type of the new key (rsa/ecc). Valid inputs are:
  - rsa: RSA key type is selected.
  - ecc: ECC key type is selected.
- `-kl`: Enter the length of the new key in bits:
  - If `-kt` is “rsa”, the supported key lengths (in bits) are 2048, 3072, and 4096.
  - If `-kt` is “ecc”, the supported key lengths are p256, p384, and p521.
- `-md`: Enter the hashing function:
  - If `-ver` is 'a' (AHAB), the supported message digests are sha256, sha384, and sha512.
  - If `-ver` is '4' (HAB), the message digest is fixed to sha256.
- `-duration`: Enter the certificate duration (in years).
- `-srk`: Choose whether it is an SRK key. Valid inputs are:
  - y: A new SRK key is generated.
  - n: A new CSF/IMG/SGK key is generated.
- `-srk-ca`: Specify whether you want the SRK certificate to have the CA flag set. Valid inputs are:
  - y: The SRK certificate has the CA flag set.
  - n: The SRK certificate does not have the CA flag set.
- `-signing-key`: Enter the signing key filename (including path information):
  - If it is an SRK key, enter the CA key filename based on whether the SRK is a CA certificate or user certificate.
  - If it is a CSF/IMG/SGK key, enter the signing SRK key filename.
- `-signing-crt`: Enter the signing certificate filename (including path information):
  - If it is an SRK key, enter the CA certificate filename based on whether the SRK is a CA certificate or user certificate.
  - If it is a CSF/IMG/SGK certificate, enter the signing SRK certificate filename.

To learn more about these options, see Section 4.1.5.1.

Note: *Before running the `add_key.sh`/`add_key.bat` script, generate a HABv4 PKI tree to avoid getting errors.*

All information provided in this document is subject to legal disclaimers.

© 2024 NXP B.V. All rights reserved.

4.2 Generating AHAB keys and certificates

This section covers the key and certificate generation process for AHAB.

Note: *When using the encrypted boot feature, digital signatures are still required.*

The data structures used by ROM and AHAB cannot be encrypted but they must be signed using a valid digital signature.

This section provides examples specifically for Security Controller (SECO) enabled devices. If you are using ELE-enabled devices, refer to the srktool command-line help for guidance on the number of supported eFuses.

4.2.1 AHAB PKI tree

To generate the tree structure for AHAB, use one of the following scripts (based on the host type):

- The `ahab_pki_tree.sh` script file for Linux
- The `ahab_pki_tree.bat` batch file for Windows

The script generates an AHAB PKI tree in the `/keys` directory of the NXP reference CST, as shown in Figure 11.

To generate the tree structure for AHAB with PQC support, use one of the following tools (based on the host type):

- The `ahab_pki_tree` tool file for Linux
- The `ahab_pki_tree.exe` tool file for Windows

The tool generates an AHAB PKI tree in the current working directory, as shown in Figure 11.

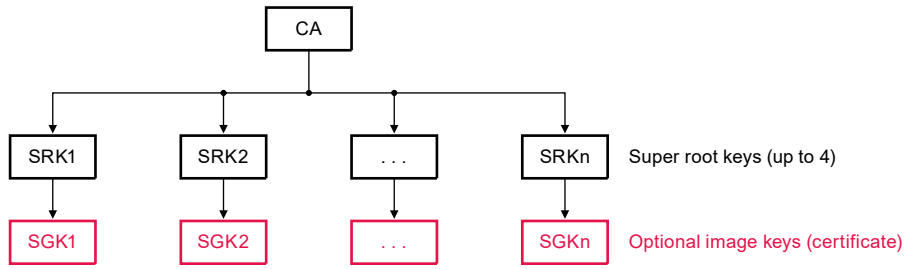

Figure 11. AHAB PKI tree

An AHAB PKI tree consists of the following keys and certificates:

- CA key: It is the top most key and is only used for signing SRK certificates.
- SRK: It is the root key for AHAB code signing keys. The cryptographic hash of a table of SRKs is burned to a set of one-time programmable eFuses to establish a root of trust. Only one of the SRKs can be selected from the table for use with the NXP processor. The SRK selection is done based on a parameter in the Install SRK CSF command (see Section 6.2.2). The SRK can be used only for signing the certificate data of the subordinate keys.
- SGK: It is a subordinate key of the SRK key and is used to verify signatures across product software.
  Note: *If the SRK keys do not have the CA flag set, the SGK keys are not generated.*

The `ahab_pki_tree` tool generates a basic tree with up to four SRKs. For each SRK, one SGK key is also generated. Using a separate tool, more keys can be added to the tree later. Moreover, you can replace the OpenSSL and the `ahab_pki_tree` tool with an alternative key generation solution. The process of creating such a solution is outside the scope of this document.

If you use an alternative key generation scheme, the scheme must follow these constraints:

All information provided in this document is subject to legal disclaimers.

© 2024 NXP B.V. All rights reserved.

- Keys must be in the PKCS#8 format.
- Certificates must be in the X.509 format and they must follow the certificate profile specified by AHAB. Keys and certificates must follow the file naming convention specified in Section 4.

Some ELE devices have restrictions on key size and hash algorithms. Table 4 summarizes the valid ELE - AHAB algorithm combination for various NXP i.MX family devices.

Table 4. ELE - AHAB algorithm support

Device	Algorithms		
	PQC	ECDSA	RSA-PSS
i.MX 8ULP	Not supported	Hash algorithms associated with elliptic curve length: • secp256 + sha2_sha256 • secp384 + sha2_sha384 • secp521 + sha2_sha512	Not supported
i.MX 93 / i.MX 91	Not supported	Hash algorithms associated with elliptic curve length: • secp256 + sha2_sha256 • secp384 + sha2_sha384 • secp521 + sha2_sha512	Key sizes: • 2k • 3k • 4k Salt length must be equal to hash size. All sha2 algorithms are supported, regardless of key size.
Other i.MX 9 family devices	Dilithium 3 or ML-DSA-65	Hash algorithms associated with key sizes: • secp256 • secp384 • secp521 All sha2, sha3, SHAKE128_256, and SHAKE256_512 algorithms are supported, regardless of key size.	Key sizes: • 2k • 3k • 4k Salt length must be equal to hash size. All sha2, sha3, SHAKE128_256, and SHAKE256_512 algorithms are supported, regardless of key size.

SHA-3 and SHAKE digest algorithm support

Compatibility with the SHA-3 or SHAKE digest algorithm depends on the cryptographic standard being used. Therefore, choosing the key type and hash type combination becomes very crucial. Consider the following points while choosing a key type - hash type combination:

- RSA has fewer restrictions on hash functions. Therefore, for RSA (using PKCS#1 v1.5 padding), fixed-length hashes from both SHA-2 (for example, sha256 and sha512) and SHA-3 (for example, sha3-256) are compatible.
- RSA-PSS requires the mask generation function (MGF1) to be parameterized that is generally limited to SHA-1 and SHA-2 hashes in libraries, such as OpenSSL. Due to the lack of support in MGF1, RSA-PSS becomes incompatible with SHAKE hashes, even if it is fixed to a specific length.
- For ECC, fixed-length SHA-2 and SHA-3 hashes are compatible. However, SHAKE hashes (even with fixed lengths) are typically unsupported due to similar reasons as RSA-PSS, relying on standard padding mechanisms that do not include SHAKE.
- In contrast, Dilithium and ML-DSA are designed to work with both fixed-length SHA-3 hashes and extendable output functions (XOFs), such as SHAKE128 and SHAKE256. Their internal structures can accommodate these hashes without relying on MGF1.
- For Hybrid, both components must support the hash function.

All information provided in this document is subject to legal disclaimers.

© 2024 NXP B.V. All rights reserved.

4.2.2 Running `ahab_pki_tree` script example

The following are the common steps to generate an AHAB PKI tree in Interactive or CLI mode for Linux. The steps for Windows are similar, with a major difference that in Windows, `ahab_pki_tree.bat` is used in place of `ahab_pki_tree.sh`:

1. Run the `cd` command:
   ```
   cd <CST Installation Path>/keys
   ```
2. Using any text editor, create a file with the name `serial` in the `/keys` directory and add a number (for example, 12345678) to it. OpenSSL uses this number as the initial certificate serial number.
3. Using any text editor, create a file with the name `key_pass.txt` in the `/keys` directory. Use this file to store your pass phrase to protect the AHAB code signing private keys. Write the pass phrase twice (in the first and second lines) in the file, for example:
   my_pass_phrase
   my_pass_phrase
   Note: *Before running the `ahab_pki_tree.sh` script, generate the `serial` and `key_pass.txt` files to avoid facing issues in generating the PKI tree.*
   CAUTION: *Ensure that the pass phrase for the private keys is protected because impacted keys cannot be used to sign the code.*
   Note: *OpenSSL enforces a minimum character length (4) for a pass phrase.*
4. Before running the `ahab_pki_tree.sh` script, include OpenSSL in your search path:
   ```
   > openssl version
   ```

4.2.2.1 Running `ahab_pki_tree` script in Interactive mode

Running the `ahab_pki_tree.sh` script (for Linux) or the `ahab_pki_tree.bat` script (for Windows) in Interactive mode prompts you with the following questions:

- Do you want to use an existing CA key (y/n)?
  - Choose 'n' unless you already have an existing CA key.
  - If you choose 'y', the script asks you to provide the filenames (including file paths) of the CA key and corresponding CA public key certificate.
- Select the key type (possible values: rsa, rsa-pss, ecc)?
  - It indicates the type of the keys in the tree.
  - If you choose "ecc", the script asks you to provide the elliptic curve (EC) to be used. For AHAB, the supported elliptic curves are P-256, P-384, and P-521.
  - If you choose "rsa" or "rsa-pss", the script asks you to enter the length in bits for the RSA keys in the tree. For AHAB, the supported RSA key lengths are 2048 bits, 3072 bits, and 4096 bits.
  - All keys in the tree are generated with the same length.
- Enter the digest algorithm to use:
  - It is the digest algorithm used to create the keys.
  - Allowed digest algorithms are sha256, sha384, and sha512.
- Enter the PKI tree duration (in years):
  - It defines the validity period of the corresponding certificates.
- Do you want the SRK certificates to have the CA flag set?
  - Answer ‘y’ for a tree with certificates as defined by the AHAB architecture.

Code listing "Example usage of `ahab_pki_tree.sh` script" illustrates the use of the `ahab_pki_tree.sh` script.

All information provided in this document is subject to legal disclaimers.

© 2024 NXP B.V. All rights reserved.

Code listing: Example usage of `ahab_pki_tree.sh` script

```
$ ./ahab_pki_tree.sh
...
<snip>
...
Do you want to use an existing CA key (y/n)?: n

Key type options (confirm targeted device supports desired key type):
Select the key type (possible values: rsa, rsa-pss, ecc)?: ecc
Enter length for elliptic curve to be used for PKI tree:
Possible values p256, p384, p521:  p384
Enter the digest algorithm to use: sha384
Enter PKI tree duration (years): 10
Do you want the SRK certificates to have the CA flag set? (y/n)?: n

++++++++++++++++++++++++++++++++++++
+ Generating CA key and certificate +
++++++++++++++++++++++++++++++++++++

Generating an EC private key
writing new private key to 'temp_ca.pem'
-----

++++++++++++++++++++++++++++++++++++++++
+ Generating SRK key and certificate 1 +
++++++++++++++++++++++++++++++++++++++++

read EC key
writing EC key
Using configuration from ../ca/openssl.cnf
Check that the request matches the signature
Signature ok
The Subject's Distinguished Name is as follows
commonName            :ASN.1 12:'SRK1_sha384_secp384r1_v3_usr'
Certificate is to be certified until Dec 12 22:09:07 2033 GMT (3650 days)

Write out database with 1 new entries
Data Base Updated
...
<snip>
...
```

4.2.2.2 Running `ahab_pki_tree` script in CLI mode

Running the `ahab_pki_tree.sh` script (for Linux) or the `ahab_pki_tree.bat` script (for Windows) in CLI mode can be helpful when automation is needed. You can run the `ahab_pki_tree` script in CLI mode as follows:

```
./ahab4_pki_tree.sh -existing-ca <y/n> [-ca-key <CA key name> -ca-cert <CA cert
 name>]
-use-ecc <y/n> -kl <ECC/RSA Key Length> -da <digest algorithm>
-duration <years> -srk-ca <y/n>
```

where:

All information provided in this document is subject to legal disclaimers.

© 2024 NXP B.V. All rights reserved.

- `-existing-ca`: Choose whether to use an existing CA key. Valid inputs are:
  - y: Provide `-ca-key` with CA key filename and `-ca-cert` with CA public key certificate filename (including path information).
  - n: The existing CA key is not selected.
- `-use-ecc`: Choose whether to use Elliptic Curve Cryptography (ECC) or RSA. Valid inputs are:
  - y: ECC keys are generated for the PKI tree.
  - n: RSA keys are generated for the PKI tree.
- `-kl`: Enter the key length for the key type selected:
  - If `-use-ecc` is 'y', provide `-kl` with the elliptic curve length for the PKI tree. Possible values are p256, p384, and p521.
  - If `-use-ecc` is 'n', provide `-kl` with the RSA length in bits for the PKI tree. For AHAB, possible values (in bits) are 2048, 3072, and 4096.
- `-da`: Enter the digest algorithm:
  - Valid inputs include sha256, sha384, and sha512.
- `-duration`: Enter the PKI tree duration (in years).
- `-num-srk`: Enter up to four SRKs (1 – 4).
- `-srk-ca`: Specify whether you want the SRK certificate to have the CA flag set. Valid inputs are:
  - y: A standard PKI tree is created.
  - n: A fast authentication PKI tree is created.

For more information on these options, see Section 4.2.2.1.

At this point, the script generates the SRK and SGK keys and certificates in the `/keys` and `/crts` directories, respectively. The generated keys exist in the PKCS#8 format (see Table 24 for more details) in both PEM and DER forms. The generated certificates are in the X.509 format (see Table 24 for more details) in both PEM and DER forms. The CST tool accepts the key and certificate files in either PEM or DER form.

After the completion of the OpenSSL key and certificate generation process, several files are left over, including `12345678.pem`, `serial.old`, and `index.txt.attr`.

At this point, you should have all the information related to the keys and certificates for signing an image using AHAB.

4.2.3 Running `ahab_pki_tree` tool with PQC support example

This section describes how to generate an AHAB PKI tree in Interactive or CLI mode for Linux. The steps for Windows are similar, with a major difference that in Windows, the `ahab_pki_tree.exe` file is used in place of the `ahab_pki_tree` tool.

The `ahab_pki_tree` tool generates keys and certificates in the `keys` and `crts` directories, respectively, under the current working directory. For easier invocation, you can add `ahab_pki_tree` to the search path of your host computer.

The tool uses a serial file to read the certificate serial number. If the file does not exist, it creates a new file with an initial serial number 12345678.

Note: *Update the serial numbers for all final production devices.*

Additionally, the tool reads the pass phrase for protecting the AHAB code signing private keys from a file, called `key_pass.txt`. If the file does not exist, the tool creates a new file with the default pass phrase "test" for protecting the private keys.

CAUTION: *Ensure that the pass phrase for the private keys is protected because impacted keys cannot be used to sign the code.*

Note: *OpenSSL enforces a minimum character length of 4 for a pass phrase.*

All information provided in this document is subject to legal disclaimers.

© 2024 NXP B.V. All rights reserved.

4.2.3.1 Running `ahab_pki_tree` tool in Interactive mode

Running the `ahab_pki_tree` tool (for Linux) or the `ahab_pki_tree.exe` tool (for Windows) in Interactive mode prompts you with the following questions:

- Do you want to use an existing CA key (y/n)?
  - Choose 'n' unless you already have an existing CA key.
  - If you choose 'y', the tool asks you to provide the filenames (including file paths) of the CA key and corresponding CA public key certificate.
- Select the key type (possible values: rsa, rsa-pss, ecc, dilithium, ml-dsa, hybrid)?
  - It indicates the type of the keys in the tree.
  - If you choose "dilithium", the tool asks you to provide the parameter set for using the Dilithium digital signature scheme. For the Dilithium scheme supporting AHAB, the supported parameter is 3.
  - If you choose "hybrid", the tool asks you to enter the classical/PQC algorithm combination. For the Dilithium 3 digital signature scheme supporting AHAB, the supported algorithm combination is p384_dilithium3.
  - All keys in the tree are generated with the same length.
- Enter the digest algorithm to use:
  - It is the digest algorithm used to create the keys.
  - Allowed digest algorithms are sha256, sha384, and sha512.
- Enter the PKI tree duration (in years):
  - It defines the validity period of the corresponding certificates.
- Do you want the SRK certificates to have the CA flag set?
  - Answer 'y' for a tree with certificates as defined by the AHAB architecture.

Code listing "Example usage of `ahab_pki_tree.sh` tool" illustrates the use of the `ahab_pki_tree` tool.

Code listing: Example usage of `ahab_pki_tree` tool

```
$ ./ahab_pki_tree
...
<snip>
...
Do you want to use an existing CA key (y/n)?: n

Select the key type (possible values: rsa, rsa-pss, ecc, dilithium, ml-dsa,
hybrid):  hybrid
Enter classical/pqc algorithm combinations (Possible values p384_dilithium3,
p384_mldsa65, p521_dilithium5, p521_mldsa87):  p384_dilithium3
Enter the digest algorithm to use (Possible values: sha256, sha384, sha512,
sha3-256, sha3-384, sha3-512):  sha384
Enter PKI tree duration (years): 10

Do you want the SRK certificates to have the CA flag set? (y/n)?: n
++++++++++++++++++++++++++++++++++++
+ Generating CA key and certificate +
++++++++++++++++++++++++++++++++++++

++++++++++++++++++++++++++++++++++++++
+ Generating SRK key and certificate 1 +
++++++++++++++++++++++++++++++++++++++

++++++++++++++++++++++++++++++++++++++
+ Generating SRK key and certificate 2 +
++++++++++++++++++++++++++++++++++++++

++++++++++++++++++++++++++++++++++++++
```


All information provided in this document is subject to legal disclaimers.

© 2024 NXP B.V. All rights reserved.

```
+ Generating SRK key and certificate 3 +
++++++++++++++++++++++++++++++++++++

++++++++++++++++++++++++++++++++++++
+ Generating SRK key and certificate 4 +
++++++++++++++++++++++++++++++++++++
```

4.2.3.2 Running `ahab_pki_tree` tool in CLI mode

Running the `ahab_pki_tree` tool (for Linux) or the `ahab_pki_tree.exe` tool (for Windows) in CLI mode can be helpful when automation is needed. You can run the `ahab_pki_tree` tool in CLI mode as follows:

```
./ahab_pki_tree -existing-ca <y/n> [-ca-key <CA key name> -ca-cert <CA cert
 name>]
-kt <Key Type> -kl <Key Length> -da <digest algorithm>
-duration <years> -srk-ca <y/n>
```

where:

- `-existing-ca`: Choose whether to use an existing CA key. Valid inputs are:
  - y: Provide `-ca-key` with CA key filename and `-ca-cert` with CA public key certificate filename (including path information).
  - n: The existing CA key is not selected.
- `-kt`: Enter the key type. Possible values are rsa, rsa-pss, ecc, dilithium, ml-dsa, and hybrid.
- `-kl`: Enter the key length for the key type selected:
  - If `-kt` is “rsa”, the supported key lengths are 2048, 3072, and 4096.
  - If `-kt` is “rsa-pss”, the supported key lengths are 2048, 3072, and 4096.
  - If `-kt` is “ecc”, the supported key lengths are p256, p384, and p521.
  - If `-kt` is “dilithium”, the supported parameter set is: 3, 5
  - If `-kt` is “ml-dsa”, the supported parameter set is: 65, 87
  - If `-kt` is “hybrid”, the supported algorithm combinations are p384_dilithium3, p384_mldsa65, p521_dilithium5, and p521_mldsa87.
- `-da`: Enter the digest algorithm:
  - Valid inputs include sha256, sha384, sha512, sha3-256, sha3-384, sha3-512, shake128, and shake256.
- `-duration`: Enter the PKI tree duration (in years).
- `-srk-ca`: Specify whether you want the SRK certificate to have the CA flag set. Valid inputs are:
  - y: A standard PKI tree is created.
  - n: A fast authentication PKI tree is created.

Code listing "Example usage of `ahab_pki_tree.sh` tool in CLI mode" illustrates the use of the `ahab_pki_tree` tool.

Code listing: Example usage of `ahab_pki_tree` tool in CLI mode

```
$ ./ahab_pki_tree -existing-ca n -kt dilithium -kl 3 -da sha384 -duration 10 -
srk-ca n...
<snip>
++++++++++++++++++++++++++++++++++++
+ Generating CA key and certificate +
++++++++++++++++++++++++++++++++++++

++++++++++++++++++++++++++++++++++++
+ Generating SRK key and certificate 1 +
++++++++++++++++++++++++++++++++++++
```


All information provided in this document is subject to legal disclaimers.

© 2024 NXP B.V. All rights reserved.

```
++++++++++++++++++++++++++++++++++++
+ Generating SRK key and certificate 2 +
++++++++++++++++++++++++++++++++++++

++++++++++++++++++++++++++++++++++++
+ Generating SRK key and certificate 3 +
++++++++++++++++++++++++++++++++++++

++++++++++++++++++++++++++++++++++++
+ Generating SRK key and certificate 4 +
++++++++++++++++++++++++++++++++++++
```

At this point, the tool generates the SRK and SGK keys and certificates in the `/keys` and `/crts` directories, respectively. The generated keys exist in the PKCS#8 format (see Table 24 for more details) in both PEM and DER forms. The generated certificates are in the X.509 format (see Table 24 for more details) in both PEM and DER forms. The CST tool accepts the key and certificate files in either PEM or DER form.

At this point, you should have all the information related to the keys and certificates for signing an image using AHAB.

4.2.4 Generating AHAB SRK tables and eFuse hash

The previous section (Section 4.2.2) explained the steps to generate the keys and certificates for an AHAB PKI tree. The next step is to generate AHAB SRK tables and corresponding hash values for burning to eFuses on the SoC.

In AHAB, the following two types of container formats are supported:

- Legacy container format
- Hybrid container format (i.MX 95 B0)

Note: *Contact NXP customer support or sales representative for devices supporting the hybrid container format.*

In the hybrid container format, two SRK tables are available. The first table is used for classical cryptographic algorithms, such as RSA, RSA-PSS, and ECDSA, as well as PQC algorithms, such as Dilithium and ML-DSA. In the hybrid container format, the second SRK table is used dedicatedly for PQC algorithms. The two SRK tables are encapsulated in an SRK table array.

When using the hybrid container format, a separate SRK hash must be generated for each table:

- `SRK_HASH` for the classical/PQC table
- `SRKH_PQC` for the dedicated PQC table

In AHAB, four SRKs are included in a signed image, though only one can be used. The SRKs are kept in a table, and one of the SRKs is selected during the boot process. The Install SRK CSF command (see Section 6.2.2) selects an SRK from the table that is used to establish the root of trust.

Any of the SRKs can be selected from the table, without changing the `SRK_HASH` values burned into eFuses on the SoC. It is useful on NXP processors where multiple eFuses are available for SRK revocation. If one or more SRKs from an SRK table are compromised, you can burn eFuses corresponding to the compromised SRKs, making them unusable. The AHAB code ensures that these SRKs are not used again. The next SRK in the table can be used to sign new images.

Up to four SRKs can be placed in an SRK table. All of them can be revoked.

Note: *In the hybrid container format, the SRK selection and SRK revoke mask operations target both normal and PQC keys. For example, when selecting SRK0, the first SRK of the first SRK table and the first SRK of the second SRK table are used. SRK index mix-match is not allowed.*

All information provided in this document is subject to legal disclaimers.

© 2024 NXP B.V. All rights reserved.

SRK tables are generated using `srktool`. Using the four SRKs created in Section 4.2.2.2, an SRK table can be created from the `/crts` directory, as shown in Code listing "SRK table and eFuse generation example".

Warning: *Do not use example values as is. Ensure that all final products are configured with unique production values and fuses.*

Code listing: SRK table and eFuse generation example

```
$ ../linux64/bin/srktool -a -s sha384 -t SRK_1_2_3_4_table.bin -e
 SRK_1_2_3_4_fuse.bin -f 1 -c ./SRK1_sha384_secp384r1_v3_usr_crt.pem,./
SRK2_sha384_secp384r1_v3_usr_crt.pem,./SRK3_sha384_secp384r1_v3_usr_crt.pem,./
SRK4_sha384_secp384r1_v3_usr_crt.pem
Number of certificates    = 4
SRK table binary filename = SRK_1_2_3_4_table.bin
SRK Fuse binary filename  = SRK_1_2_3_4_fuse.bin
SRK Fuse binary dump:
SRK HASH[0] = 0xA83170EF
SRK HASH[1] = 0xFA1D5EA8
SRK HASH[2] = 0xE3A2C737
SRK HASH[3] = 0xAD1D0241
SRK HASH[4] = 0x6246BE44
SRK HASH[5] = 0x75439F14
SRK HASH[6] = 0x9F65FC0B
SRK HASH[7] = 0x6DAC9B80
SRK HASH[8] = 0x9481C935
SRK HASH[9] = 0x8C6CC5EC
SRK HASH[10] = 0x9104B9E5
SRK HASH[11] = 0x5F97C971
SRK HASH[12] = 0x0DA8DDA5
SRK HASH[13] = 0xAC21273D
SRK HASH[14] = 0x0FCE73F7
SRK HASH[15] = 0x3FC9ACBA
```

In this example:

- All four SRKs are included in the table.
- The signature hash algorithm used for signing is SHA-384 (`-s` option).
- The hash value to be fused is generated with 32 bits of eFuse data per word. It is true for SECO-based devices (for example, i.MX 8QXP/QM) that use the SHA-512 hash algorithm for eFuses as well as for ELE-based devices (for example, i.MX 8ULP, i.MX 93, and i.MX 95 A0) that use the SHA-256 hash algorithm for eFuses. However, some NXP processors require the hash value to be generated with 8 bits of eFuse data per word. In that case, use the `-f 0` option.
- The hash result is in the resulting `SRK_1_2_3_4_fuse.bin` file.

Code listing "SRK table array and eFuse generation from hybrid keys example" shows an example SRK table array and explains how to generate eFuses from hybrid keys.

Code listing: SRK table array and eFuse generation from hybrid keys example

```
$ ../linux64/bin/srktool -a 2 -s sha384 -t SRK_1_2_3_4_tables.bin -e
 SRK_1_2_3_4_fuses.bin -f 1 -c ./SRK1_sha384_p384_dilithium3_v3_usr_crt.pem,./
SRK2_sha384_p384_dilithium3_v3_usr_crt.pem,./
SRK3_sha384_p384_dilithium3_v3_usr_crt.pem,./
 SRK4_sha384_p384_dilithium3_v3_usr_crt.pem
Number of certificates = 4
SRK table binary filename = SRK_1_2_3_4_tables.bin
SRK Fuse binary filename = SRK_1_2_3_4_fuses.bin
SRK Fuse binary dump:
SRK HASH[0] = 0x9EAE2E44
```


All information provided in this document is subject to legal disclaimers.

© 2024 NXP B.V. All rights reserved.

```
SRK HASH[1] = 0xE11AD5B7
SRK HASH[2] = 0x5C2799B3
SRK HASH[3] = 0x6D9E8732
SRK HASH[4] = 0xD422A125
SRK HASH[5] = 0xD731BC2B
SRK HASH[6] = 0x97A74470
SRK HASH[7] = 0xEA71800A
SRK HASH[8] = 0xC0EB92A7
SRK HASH[9] = 0x49861BD4
SRK HASH[10] = 0x3DD22111
SRK HASH[11] = 0x6C2D487C
SRK HASH[12] = 0x9E6FBB52
SRK HASH[13] = 0xD3DB0AA3
SRK HASH[14] = 0x0B7AF95B
SRK HASH[15] = 0x05881C85
SRKH_PQC[0] = 0x95FBFCEC
SRKH_PQC[1] = 0x6B764B02
SRKH_PQC[2] = 0x5DDEBA1C
SRKH_PQC[3] = 0x6ED213DB
SRKH_PQC[4] = 0x8391549E
SRKH_PQC[5] = 0xAB30B4F3
SRKH_PQC[6] = 0x897C19D1
SRKH_PQC[7] = 0x5AB23DEA
SRKH_PQC[8] = 0x587C5179
SRKH_PQC[9] = 0xA3108831
SRKH_PQC[10] = 0x395869DE
SRKH_PQC[11] = 0xC13ABF64
SRKH_PQC[12] = 0x3D1B8221
SRKH_PQC[13] = 0xC229F760
SRKH_PQC[14] = 0xA8C1C37A
SRKH_PQC[15] = 0x4C84C288
```

In this example:

- The SRK table array contains two SRK tables.
- All four SRKs are included in each table.
- The hybrid container format is provided (`-a 2` option).
- The signature hash algorithm used for signing is SHA-384 (`-s` option).
- The hash value to be fused is generated with 32 bits of eFuse data per word. Devices supporting hybrid container format (for example, i.MX 95 B0) use the SHA-512 hash algorithm for eFuses.
- The hash result (concatenation of the two hashes from the two tables) is in the resulting `SRK_1_2_3_4_fuses.bin` file.

Note: *Do not add a whitespace between the ',' when specifying the SRKs in the `-c` or `--certs` option. Adding a whitespace results in all certificates specified after the first whitespace not to be included in the table. It also causes an execution error of `srktool`.*

4.2.5 Programming SRK hash value to eFuses

The previous section provided details on how to generate the SRK tables and the corresponding eFuse data. This section covers how to program the SRK hash value. The hash value stored in the eFuse file is burned to the `SRK_HASH` and/or `SRKH_PQC` eFuse field on the SoC supporting AHAB.

The `SRK_1_2_3_4_fuse.bin` file from Code listing "SRK table and eFuse generation example in Section 4.2.4 has the following contents:

```
ef7031a8a85e1dfa37c7a2e341021dad44be4662149f43750bfc659f809bac6d35c98194ecc56c8ce5b9049171c9975fa5dda80d3d2721acf773ce0fbaacc93f
```


All information provided in this document is subject to legal disclaimers.

© 2024 NXP B.V. All rights reserved.

The corresponding hexadecimal dump of the eFuse file is shown below:

```
$ hexdump -C SRK_1_2_3_4_fuse.bin
00000000  ef 70 31 a8 a8 5e 1d fa  37 c7 a2 e3 41 02 1d ad  |.p1..^..7...A...|
00000010  44 be 46 62 14 9f 43 75  0b fc 65 9f 80 9b ac 6d  |D.Fb..Cu..e....m|
00000020  35 c9 81 94 ec c5 6c 8c  e5 b9 04 91 71 c9 97 5f  |5.....l.....q.._|
00000030  a5 dd a8 0d 3d 27 21 ac  f7 73 ce 0f ba ac c9 3f  |....='!..s.....?|
```

This hash value must be burned into the SoC eFuses in the following order (the first word to the first fuse row index):

```
$ hexdump -e '/4 "0x"' -e '/4 "%X""\n"' SRK_1_2_3_4_fuse.bin
0xA83170EF
0xFA1D5EA8
0xE3A2C737
0xAD1D0241
0x6246BE44
0x75439F14
0x9F65FC0B
0x6DAC9B80
0x9481C935
0x8C6CC5EC
0x9104B9E5
0x5F97C971
0xDA8DDA5
0xAC21273D
0xFCE73F7
0x3FC9ACBA
```

To know the location of the `SRK_HASH` field, refer to the fuse map of the NXP processor in use.

The `SRK_1_2_3_4_fuses.bin` file from Code listing "SRK table array and eFuse generation from hybrid keys example" in Section 4.2.4 has the following hexadecimal dump:

```
$ echo "-- SRK_HASH --" && hexdump -e '/4 "0x"' -e '/4 "%X""\n"' fuses.bin | awk
 'NR==16 {print "-- SRKH_PQC --"} {print}'

-- SRK_HASH --
0x9EAE2E44
0xE11AD5B7
0x5C2799B3
0x6D9E8732
0xD422A125
0xD731BC2B
0x97A74470
0xEA71800A
0xC0EB92A7
0x49861BD4
0x3DD22111
0x6C2D487C
0x9E6FBB52
0xD3DB0AA3
0xB7AF95B
-- SRKH_PQC --
0x5881C85
0x95FBFCEC
0x6B764B02
0x5DDEBA1C
0x6ED213DB
```


All information provided in this document is subject to legal disclaimers.

© 2024 NXP B.V. All rights reserved.

```
0x8391549E
0xAB30B4F3
0x897C19D1
0x5AB23DEA
0x587C5179
0xA3108831
0x395869DE
0xC13ABF64
0x3D1B8221
0xC229F760
0xA8C1C37A
0x4C84C288
```

These hash values must be burned into the corresponding SoC eFuses in the order they are displayed (the first word goes to the first fuse row index).

To know the locations of the `SRK_HASH` and `SRKH_PQC` fields, refer to the fuse map of the NXP processor in use.

4.2.6 Adding a key to an AHAB PKI tree

A new key can be added to an existing AHAB PKI tree by running one of the following scripts (based on the host type):

- The `add_key.sh` script file for Linux
- The `add_key.bat` batch file for Windows

Running the `add_key.sh`/`add_key.bat` script in Interactive mode prompts you with the following questions:

- Which version of HAB/AHAB do you want to generate the key for (3/4/a)?
  - Enter 'a' for AHAB.
- Enter new key name (for example, SRK5):
  - Specify a name for the new key, for example, SRK2 or SGK3.
- Enter new key type (ecc / rsa / rsa-pss):
  - Specify the type of the new key: ECC, RSA, or RSA-PSS
- Enter new key length (in bits):
  - Specify the length of the new key in bits that matches the length of the signing key.
- Enter new message digest:
  - Specify the digest of the key signature.
- Enter certificate duration (in years):
  - Specify the validity period for the certificate to be generated.
- Is this an SRK key?
  - If you are generating a new SRK key, enter ‘y’; otherwise, enter ‘n’.
  - If you enter ‘y’, you are prompted with “Do you want the SRK to have the CA flag set?”. Enter ‘y’ if you are generating an SRK with the CA flag set.
  - If you enter ‘n’, it implies that you are generating a new SGK key.
- Enter `<key type>` signing key name:
  - If you are generating a new SRK key, `<key type>` is CA. Enter the path and filename of the CA key in the `/keys` directory that you want to use for generating the SRK key.
  - If you are generating a new SGK key, `<key type>` is SRK. Enter the path and filename of the SRK in the `/keys` directory that you want to use for generating the SGK key.
- Enter `<cert type>` signing certificate name:
  - If you are generating a new SRK certificate, the `<cert type>` is CA. Enter the path and filename of the CA certificate in the `/crts` directory that you want to use for generating the SRK certificate.

All information provided in this document is subject to legal disclaimers.

© 2024 NXP B.V. All rights reserved.

– If you are generating a new SGK certificate, `<cert type>` is SRK. Enter the path and filename of the SRK certificate in the `/certs` directory that you want to use for generating the SGK certificate.

Using the keys generated in Section 4.2.2, a new SRK key can be added to the PKI tree, as shown in Code listing "Adding a new SRK to an AHAB PKI tree example".

Code listing: Adding a new SRK to an AHAB PKI tree example

```
$ ./add_key.sh
Which version of HAB/AHAB do you want to generate the key for (4 = HAB4 / a =
 AHAB)?: a
Enter new key name (e.g. SRK5):
Enter new key type (ecc / rsa / rsa-pss): ecc
Enter new key length (p256 / p384 / p521): p384
Enter new message digest (sha256, sha384, sha512): sha384
Enter certificate duration (years): 10
Is this an SRK key?: y
Do you want the SRK to have the CA flag set (y/n)?: n
Enter CA signing key name: CA1_sha384_secp384r1_v3_ca_key.pem
Enter CA signing certificate name: ../crts/CA1_sha384_secp384r1_v3_ca_crt.pem
read EC key
writing EC key
Using configuration from ../ca/openssl.cnf
Check that the request matches the signature
Signature ok
The Subject's Distinguished Name is as follows
commonName            :ASN.1 12:'_sha384_secp384r1_v3_usr'
Certificate is to be certified until Dec 12 22:25:42 2033 GMT (3650 days)

Write out database with 1 new entries
Data Base Updated
```

Note: *Before running the `add_key.sh`/`add_key.bat` script, generate an AHAB PKI tree to avoid getting errors.*

Note: *For PQC support, the `add_key` binary mimics the `add_key` scripts with support of PQC and hybrid algorithms.*

All information provided in this document is subject to legal disclaimers.

© 2024 NXP B.V. All rights reserved.

5 CST usage

This section describes how to use the CST and other tools in the release package.

5.1 Code signing tool (CST)

The `cst` tool in the release package is the main application used to generate binary CSF data using input CSF description files passed as standard input. You can run the CST from any location by providing the correct absolute or relative path to the location. The paths to certificate and image files inside CSF can be either relative to the current working directory location or as absolute paths.

Due to a limitation in current CST implementation, the CST must be run from a directory at the same level as `<Installation path>/keys`. For example, you can run the CST from `<Installation path>/product_code`, where the product code to be signed is located.

Usage:

```
cst --output <binary> [--cert <cert_file>] --input <input_csf>
        [--license] [--help]
```

Description:

```
-o, --output <binary csf>:
    Output binary CSF filename

-i, --input <csf text file>:
    Input CSF text filename

-c, --cert <public key certificate>:
    Optional, Input public key certificate to encrypt the dek

-b, --backend <ssl or pkcs11>:
    Optional, Select backend. SSL backend is the default and
    uses keys stored in the local host filesystem. The PKCS11
    backend supplies an interface to PKCS11 supported keystore.

-g, --verbose:
    Optional, displays verbose information.  No additional
    arguments are required

-l, --license:
    Optional, displays program license information.  No additional
    arguments are required

-v, --version:
    Optional, displays the version of the tool.  No additional
    arguments are required

-h, --help:
    Optional, displays usage information.  No additional
    arguments are required
```

If enclosed in double quotes, a command-line argument that specifies a file or directory can contain whitespaces. Filenames with leading or trailing whitespaces are not supported.

If an error occurs during the operation of the CST, an error message is printed to the standard output stream. As a result, the executable exits with a non-zero status.

All information provided in this document is subject to legal disclaimers.

© 2024 NXP B.V. All rights reserved.

Exit status:

- 0 if the executable succeeded
- > 0 otherwise

Cautions:

None

Pre-conditions / assumptions:

- The input CSF file must be present at the specified path.
- Certificates must be in a directory called `crts`.
- Keys must be in a directory called `keys`. The `keys` directory must be at the same level as the `crts` directory.
- Filenames for the keys and certificates must use one of the following conventions:
  - `<filename>_<type>.pem`
  - `<filename>_<type>.der`

  where:
  - `<filename>` is the root of the key/certificate filename.
  - `<type>` is `key` for keys and `crt` for certificates.
- The NXP reference CST requires a one-to-one mapping between the keys in the `/keys` directory and the certificates in the `/crts` directory. For example, a key named `SRK1_sha256_2048_65537_v3_ca_key.der` in the `/keys` directory must have a corresponding certificate `SRK1_sha256_2048_65537_v3_ca_crt.der` in the `/crts` directory.

Post conditions:

None

Examples:

- To generate the `out.bin` file from the input HABv4 CSF file, run the following command:

```
cst -o out.bin -i example.csf
```

- To print program license information, run the following command:

```
cst --license
```

- To print usage information, run the following command:

```
cst --help
```

- To generate the `out.bin` file from the input HABv4 CSF file and public key certificate file for encrypting symmetric keys, run the following command:

```
cst -o out.bin --cert dek_protection_crt.pem -i example.csf
```

5.2 SRK tool

The *srktool* is used to generate super root key (SRK) table data and SRK table array data, along with SRK hashes (for eFuses).

5.2.1 SRK tool usage for HABv4

This section describes the usage of srktool for HABv4.

All information provided in this document is subject to legal disclaimers.

© 2024 NXP B.V. All rights reserved.

Usage:

```
srktool --hab_ver <version> --table <tablefile> --efuses <efusefile>
        --digest <digestalg> --certs <srk>,%<srk>,...
        [--fuse_format <format>] [--license]
```

Description:

```
  -h, --hab_ver <version>:
      HAB Version - set to 4 for HAB4 SRK table generation

  -t, --table <tablefile>:
      Filename for output SRK table binary file

  -e, --efuses <efusefile>:
      Filename for the output SRK efuse binary file containing the SRK table
      hash

  -d, --digest <digestalg>:
      Message Digest algorithm. Only sha256 is supported

  -c, --certs <srk1>,<srk2>,...,<srk4>:
      X.509v3 certificate filenames.
        - Certificates may be either DER or PEM encoded format
        - Certificate filenames must be separated by a ','with no spaces
        - A maximum of 4 certificate filenames may be provided. Additional
          certificate names are ignored
        - Placing a % in front of a filename replaces the public
          key data in the SRK table with a corresponding hash digest

  -f, --fuse_format <format>:
      Optional, Data format of the SRK efuse binary file.  The
      format may be selected by setting <format> to either:
        - 0: 8 fuses per word, ex: 00 00 00 0a 00 00 00 01 ...
        - 1 (default): 32 fuses per word, ex: 0a 01 ff 8e

   -l, --license:
      Optional, displays program license information.  No additional
      arguments are required.

   -v, --version:
      Optional, displays the version of the tool.  No additional
      arguments are required.

   -b, --verbose:
      Optional, displays a verbose output.
```

If enclosed in double quotes, a command-line argument that specifies a file or directory can contain whitespaces. Filenames with leading or trailing whitespaces are not supported.

If an error occurs during the operation of srktool, an error message is printed to the standard output stream. As a result, the executable exits with a non-zero status.

Exit status:

- 0 if the executable succeeded
- > 0 otherwise

All information provided in this document is subject to legal disclaimers.

© 2024 NXP B.V. All rights reserved.

Using the % prefix in the `-c` option does not change the SRK fuse pattern generated but it reduces the overall size of the SRK table. However, an SRK prefixed with % cannot be selected in the Install SRK command using that SRK table.

Cautions:

None

Pre-conditions / assumptions:

None

Post conditions:

None

Examples:

- To generate an SRK table and corresponding fuse pattern using three certificates:

```
srktool --hab_ver 4 --table table.bin  --efuses fuses.bin \
                --digest sha256 \
                --certs srk1_crt.pem,srk2_crt.pem,srk3_crt.pem
```

- To generate an SRK table and corresponding fuse pattern using two certificates:
  - Using DER encoded certificate files
  - Using the optional 8 fuse bits per word for the eFuse file

```
srktool --hab_ver 4 --table table.bin  --efuses fuses.bin \
                --digest sha256 \
                --certs srk1_crt.der,srk2_crt.der\
                --fuse_format 1
```

5.2.2 SRK tool usage for AHAB

This section describes the usage of srktool for AHAB.

Usage:

```
srktool --ahab_ver <ahabversion> --table <tablefile> --efuses <efusefile>
        --sign_digest <digestalg> --certs <srk>,<srk>,...
        [--fuse_format <format>] [--license]
```

Description:

```
srktool --ahab_ver --table <tablefile> --efuses <efusefile>
        --sign_digest <digestalg> --certs <srk>,<srk>,...
        [--fuse_format <format>] [--license]

  -a, --ahab_ver <ahabversion>:
      AHAB Version - set for AHAB SRK table or SRK table array generation
  (1: legacy, 2: hybrid)

  -t, --table <tablefile>:
      Filename for output SRK table binary file

  -e, --efuses <efusefile>:
      Filename for the output SRK efuse binary file containing the SRK table
      hash

  -d, --digest <digestalg>:
```


All information provided in this document is subject to legal disclaimers.

© 2024 NXP B.V. All rights reserved.

```
      Message Digest algorithm.
          - sha512 (default): Supported in SECO-based devices (i.MX8/i.MX8x)
            and ELE-based devices (i.MX95B0)
          - sha256: Supported in ELE-based devices (i.MX8ULP, i.MX93,
            i.MX95A0, ..)

  -s, --sign_digest <digestalg>:
      Signature Digest algorithm. Either sha256, sha384, sha512, sha3-256,
 sha3-384, sha3-512, shake128, or shake256.

  -c, --certs <srk1>,<srk2>,...,<srk4>:
      X.509v3 certificate filenames.
        - Certificates may be either DER or PEM encoded format
        - Certificate filenames must be separated by a ','with no spaces
        - A maximum of 4 certificate filenames may be provided. Additional
          certificate names are ignored
  -f, --fuse_format <format>:
      Optional, Data format of the SRK efuse binary file.  The
      format may be selected by setting <format> to either:
        - 0: 8 fuses per word, ex: 00 00 00 0a 00 00 00 01 ...
        - 1 (default): 32 fuses per word, ex: 0a 01 ff 8e

   -l, --license:
      Optional, displays program license information.  No additional
      arguments are required.

   -v, --version:
      Optional, displays the version of the tool.  No additional
      arguments are required.

   -b, --verbose:
      Optional, displays a verbose output.
```

If enclosed in double quotes, a command-line argument that specifies a file or directory can contain whitespaces. Filenames with leading or trailing whitespaces are not supported.

If an error occurs during the operation of srktool, an error message is printed to the standard output stream. As a result, the executable exits with a non-zero status.

Exit status:

- 0 if the executable succeeded
- > 0 otherwise

Cautions:

None

Pre-conditions / assumptions:

None

Post conditions:

None

Examples:

- To generate an SRK table and corresponding fuse pattern:
  - Using PEM encoded certificate files

All information provided in this document is subject to legal disclaimers.

© 2024 NXP B.V. All rights reserved.

– Using the default 32 fuse bits per word for the eFuse file

```
srktool --ahab_ver --table table.bin --efuses fuses.bin \
        --sign_digest sha384 \
        --certs srk1_crt.pem,srk2_crt.pem,srk3_crt.pem,srk4_crt.pem
```

- To generate an SRK table and corresponding fuse pattern:
  – Using DER encoded certificate files
  – Using the optional 8 fuse bits per word for the eFuse file

```
srktool --ahab_ver --table table.bin --efuses fuses.bin \
        --sign_digest sha256 \
        --certs srk1_crt.der,srk2_crt.der,srk3_crt.der,srk4_crt.der \
        --fuse_format 1
```

- To generate an SRK table array and corresponding fuse pattern:

```
srktool --ahab_ver 2 --table tables.bin --efuses fuses.bin \
  --sign_digest sha384 \
  --certs srk1_crt.pem,srk2_crt.pem,srk3_crt.pem,srk4_crt.pem
```

5.3 HABv4 MAC dump tool

The *hab4_mac_dump* tool is used to dump the MAC data location and size from a HABv4 CSF binary data file. It iterates over the tags in a provided binary CSF file, looking for the position of the *HAB_TAG_MAC* tag. By doing so, it helps to retrieve the correct offset of the nonce/MAC. This information is useful in constructing a correctly signed encrypted image.

Usage:

```
hab4_mac_dump <path to csf file>
```

Example:

To dump the offset and length of the nonce/MAC on the CSF binary file of an encrypted Secondary Program Loader (SPL), run the following command:

```
hab4_mac_dump spl_encrypt_csf.bin
MAC_TAG offset: 0xa38
MAC_TAG length: 0x25
```

5.4 AHAB split container tool

The *ahab_split_container* tool is used to split the NXP-signed AHAB container into ELE and V2X containers, enabling the OEM double signing feature. This feature enforces the use of a specific ELE firmware version and a specific V2X firmware version in the OEM boot image.

Usage:

```
ahab_split_container [options] <input_file>
```

Example:

To split an i.MX 95 A0 AHAB container into ELE and V2X containers, run the following command:

```
$ ./ahab_split_container mx95a0-ahab-container.img
```


All information provided in this document is subject to legal disclaimers.

© 2024 NXP B.V. All rights reserved.

The resulting files `mx95a0-ahab-container.img.ele` and `mx95a0-ahab-container.img.v2xfh` are created in the same folder as the `mx95a0-ahab-container.img` file.

Note: *The i.MX 95 A0 is a pre-production device. The i.MX 95 B0 is anticipated to be the production version.*

5.5 AHAB signed message tool

The *ahab_signed_message* tool is used to generate signed messages for unlocking services of AHAB devices running on an OEM-closed life cycle. The generated signed message must be signed later using CST with a valid key.

Usage:

```
ahab_signed_message -t <message_template_file> -m <message_payload_file>
-o <message_output_file>
```

Templates and payloads for messages supported by SECO and ELE can be found under the examples folder on GitHub.

Example:

To generate a signed message for SECO to move the life cycle of the part to Field Return, run the following command:

```
$ ./ahab_signed_message ahab_return_lifecycle_update_template.json
ahab_return_lifecycle_update_payload.json
-o ahab_return_lifecycle_update_message.bin
```

The result containing the signed message is `ahab_return_lifecycle_update_message.bin`.

5.6 HABv4 image verifier tool

The *hab4_image_verifier* tool is used to parse and verify HABv4 images, including CSF commands.

Usage:

```
hab4_image_verifier <image> <offset>
```

Note: *The hab4_image_verifier tool is intended solely for experimentation and debugging purposes. Under any circumstance, it should not be considered as a proof or validation of a good HABv4 image. For official verification and image integrity checks, proceed with the recommended secure boot procedures, such as using the `hab_status` command as outlined in the relevant documentation.*

Example:

To parse CSF commands in an i.MX 8M boot flash image, run the following command:

```
$./hab4_image_verifier  signed_flash.bin 0
Offset: 0x0
Image Vector Table:
  Offset: 0x0
  IVT Tag: 0xD1
  IVT Version: 0
  IVT Length: 0x20
  HAB Version: 0x41
  Entry: 0x7E1000
  DCD: 0x0
  Boot Data: 0x7E0FE0
```


All information provided in this document is subject to legal disclaimers.

© 2024 NXP B.V. All rights reserved.

```
  Self: 0x7E0FC0
  CSF: 0x80CFC0
  Command Sequence File (CSF):
  Offset: 0x2C000
  CSF Tag: 0xD4
  CSF Length: 88 bytes
  HAB Version: 0x43
Command:
    Command Tag: 0xBE
    Command Length: 12 bytes
    Processing 'Install Key' command...
    Flags: 0x0
    Protocol Tag: 0x3 (SRK certificate format)
    Algorithm Tag: 0x17 (SHA-256 algorithm)
    SRK Table processed
    Source Index: 0
    Target Index: 0
    Certificate Offset: 0x58
<snip> ..
```

5.7 AHAB image verifier tool

The *ahab_image_verifier* tool is used to parse and verify AHAB containers. It supports both legacy and hybrid container formats.

Usage:

```
ahab_image_verifier <container> <offset>
```

Note: *The ahab_image_verifier tool is intended solely for experimentation and debugging purposes. Under any circumstance, it should not be considered as a proof or validation of a good AHAB container. For official verification and container integrity checks, proceed with the recommended secure boot procedures, such as using the* `ahab_status` *command as outlined in the relevant documentation.*

Example:

To parse and verify an ELE firmware container in an i.MX 95 A0 AHAB image, run the following command:

```
$ ./ahab_image_verifier mx95a0-ahab-container.img 0
Offset: 0x0
Header:
  Version: 0
  Length: 544 bytes
  Tag: Container header (0x87)
  Flags: 0x1
    SRK Set: NXP SRK
    SRK Selection: SRK 0 is used
    SRK Revoke Mask: None
SW Version: 0
  Fuse Version: 0
  Number of Images: 1
  Signature Block Offset: 0x90
<snip> ..
Signature verification successful.
```

To parse and verify a V2X firmware container in an i.MX 95 A0 AHAB image, run the following command:

```
$ ./ahab_image_verifier mx95a0-ahab-container.img 0x400
```


All information provided in this document is subject to legal disclaimers.

© 2024 NXP B.V. All rights reserved.

```
<snip> ..
Image 1:
  Offset: 0x1BC00
  Size: 53248 bytes
  Load Address: 0x6000000
  Entry Point: 0x6000000
  Image Flags: 0x19B
    Type of Image: V2X 1 FW
    Core ID: V2X 1
    Hash Type: SHA2_384
    Encrypted: No
  Reserved: 0x0
<snip> ..
Signature verification successful.
```

5.8 HABv4 log parser

The *hab_log_parser* tool is used to parse the HAB-persistent memory dumps and print the HAB events. ROM/HAB allocates certain memory region in the internal RAM (OCRAM) for HAB logs. This space is marked as reserved in the internal RAM memory map and it must not be edited. This memory region is called *HAB-persistent memory*. It contains certificates, events, and other information related to the HAB process.

The HAB-persistent memory is used by HAB to store audit logs, that is:

- Logs of various events
- Status results generated from HAB library functions

HAB uses this memory region to search through existing events to report when requested through an API call. HAB stores certificate information of the keys installed as reported in the CSF file.

Table 5 provides HAB-persistent memory information for various NXP devices.

Table 5. HAB-persistent memory information

Device	HAB-persistent memory address	Size
i.MX 6S/DL/D/Q/DP/QP/UL/ULL/SL/SLL/SX	0x904000	0xB80
i.MX 7S/D	0x9049C0	0xB80
i.MX 7ULP1 A7 B0	0x2F006840	0xB80
i.MX 7ULP1 M4 B0	0x20008040	0xB80
MSCALE 850D B0	0x9061C0	0xB80

Usage:

```
hab_log_parser [input] [output]
Input:
        -s|--sdp <device_name>: SDP mode selected with Device name required
                 <device_name>:
                        '-----imx6s
                        '-----imx6dl
                        '-----imx6q
                        '-----imx6d
                        '-----imx6qp
                        '-----imx6dp
                        '-----imx6sl
                        '-----imx6sll
                        '-----imx6sx
                        '-----imx6ul
```


All information provided in this document is subject to legal disclaimers.

© 2024 NXP B.V. All rights reserved.

```
                          '-----imx6ull
                          '-----imx7s
                          '-----imx7d
                          '-----imx7ulpa7
                          '-----imx7ulpm4
                          '-----imx8mq
        -b|--input-bin <input file>: Binary file containing a dump of HAB4
 persistent memory contents
        -a|--input-ascii <input file>: Binary file containing a dump of HAB4
 persistent memory contents
Output (Optional):
        -o|--output <output>: File of the parsed HAB4 persistent memory region
                            : If output not provided output is sent to stdout
```

Examples:

- Reading HAB log from target using serial download protocol (SDP):

```
hab_log_parser -s imx6s -o hab_log_parsed.txt
```

- Getting binary dump of HAB log SRAM locations from a file:

```
hab_log_parser -b hab4_pers.bin -o hab_log_parsed.txt
```

- Getting ASCII dump of HAB log SRAM locations from a file:

```
hab_log_parser -a hab4_pers.ascii -o hab_log_parsed.txt
```

Extracting debug information:

- Via JTAG:
  Trace32 (Lauterbach):

```
DATA.SAVE.BINARY <filename> <hab persistent memory address>--<hab persistent
 memory address + size>
```

  DSTREAM (DS5):

```
dump binary memory <filename> <hab persistent memory address> <hab persistent
 memory address + size>
```

- Via U-Boot:
  1. Dump the persistent memory region with U-Boot `md` command:

```
//for imx7ulp B0 A7 (u-boot)
    u-boot=> md.b 0x2f006840 0xb80

//for imx7ulp B0 M4 (u-boot)
    u-boot=> md.b 0x20008040 0xb80
```

  2. Copy the log and save in the `uboot_dump.txt` file.
  3. Adapt U-Boot log to ASCII format:

```
$ cat uboot_dump.txt | cut -c 11- | cut -c -48 > uboot_persistent_ascii.txt
```

5.9 HABv4 CSF parser

The *hab_csf_parser* tool is developed to help users:

- Parse the CSF binary either from a signed image or a standalone CSF binary
- Output debug data

All information provided in this document is subject to legal disclaimers.

© 2024 NXP B.V. All rights reserved.

The parsed CSF also extracts the certificates, signatures, and SRK table.

Usage:

```
hab_csf_parser [-d] [[-s <signed_image>] | [-c <csf_binary>]] options: -d|--
enable-debug --> Enable Debug information -s|--signed-image --> Input signed
 image -c|--csf-binary --> Input CSF binary
```

Note: *Only one image can be parsed at a time.*

All information provided in this document is subject to legal disclaimers.

© 2024 NXP B.V. All rights reserved.

6 CSF description language

This section describes the Command Sequence File (CSF) description language. A CSF description file is written in the CSF description language. The CST application parses and processes the CSF description file, generating a binary file that contains:

- CSF commands (only valid for HAB)
- Certificates
- Signatures. On the end-product device, the secure element interprets the signatures.

6.1 Overview

The following are the general properties of a CSF description file:

- A CSF description file is a text file containing statements, one per line.
- A backslash character '\' at the end of a line (ignoring whitespaces or comments) continues the statement to the next line.
- Blank lines are ignored.
- Comments beginning with the # character on any line are ignored.
- Multiple whitespace characters are equivalent to a single whitespace. Unless specified explicitly, keywords and parameters are separated by a whitespace. Whitespaces at the beginning or end of a line are ignored.
- Except for filenames, all keywords and parameters are case-insensitive.
- All certificate file parameters are relative to the current folder from where the CST application is being executed.
- All byte parameters are specified as integers in the range from 0 to 255. They can be specified in hexadecimal or decimal.
- All parameters that specify a filename must be enclosed in double quotes. A quoted filename can contain whitespaces. The following filenames are not supported:
  - A filename with leading or trailing whitespaces
  - A filename that contains a double quote (") as part of the filename
- The sequence of commands within the CSF description file is significant only to the following extent:
  - The Header command must precede any other command. It is valid for both HAB and AHAB. The next statements are valid only for HAB.
  - The Install SRK command must precede the Install CSFK command.
  - The Install CSFK command must precede the Authenticate CSF command.
  - Install SRK, Install CSFK, and Authenticate CSF commands must appear exactly once in a CSF description file.
  - A verification index in an Authenticate Data command must appear as a target index in an Install Key command mentioned previously.
  - Commands in the binary CSF file follow the sequence in which they appear in the CSF description file.

6.2 CSF commands

This section describes the CSF commands.

6.2.1 Header

The Header command contains:

- Data used in the CSF header
- Default values that the CST uses for other commands in the remaining CSF

All information provided in this document is subject to legal disclaimers.

© 2024 NXP B.V. All rights reserved.

Each CSF file must contain exactly one Header command that appears at the beginning of the CSF file.

Table 6 lists the Header command arguments.

Table 6. Header arguments

Argument	Description	Valid values	HABv4	AHAB
Target	Target secure element. If not specified, HAB is assumed to be the target secure element.	HAB and AHAB	O	M
Version	Version of HAB or AHAB	For HAB, the version format is 4.x, where x = 0, 1, or so on. For AHAB, the version is one of the following values: • 1: Legacy container format • 2: Hybrid container format	M	M
Mode	Mode of CST execution (to be specified only for HSM handling)	HSM	O	O
Hash Algorithm	Default hash algorithm	SHA-256	O	X
Engine	Default engine	ANY, SAHARA, RTIC, DCP, CAAM, and SW	O	X
Engine Configuration	Default engine configuration	See Table 7	O	X
Certificate Format	Default certificate format	X509	O	X
Signature Format	Default signature format	PKCS1 and CMS	O	X
Signature Size	Defines the expected size of the CMS signature. This command only works for HSM mode when the signature format is set to CMS.	The appropriate byte length needed to store a signature generated by an HSM	O	X

In Table 6,

- M = Mandatory
- O = Optional
- X = Not present

Table 7 lists the valid engine configuration values for each engine type.

Table 7. Valid engine configuration values

Engine name	Valid engine configuration values
ANY	0
SAHARA	One or more of the following values, separated by '|': • 0 • IN SWAP8 • IN SWAP16 • DSC BE816 • DSC BE832
DCP	One or more of the following values, separated by '|': • 0

All information provided in this document is subject to legal disclaimers.

© 2024 NXP B.V. All rights reserved.

Table 7. Valid engine configuration values*...continued*

Engine name	Valid engine configuration values
	• IN SWAP8 • IN SWAP32 • OUT SWAP8 • OUT SWAP32
CAAM	One or more of the following values, separated by '|': • 0 • IN SWAP8 • IN SWAP16 • OUT SWAP8 • OUT SWAP16 • DSC SWAP8 • DSC SWAP16
RTIC	One or more of the following values, separated by '|': • 0 • IN SWAP8 • IN SWAP16 • OUT SWAP8 • KEEP
SW	0

6.2.1.1 Header usage

The following code snippet shows how to use the Header command:

```
[Header]
Version = 4.1 # HAB4 example
Hash Algorithm = SHA256
Engine = Any
Engine Configuration = 0
Certificate Format = X509
Signature Format = CMS
[Header]
Target = AHAB # AHAB example
Version = 1.0
```

6.2.2 Install SRK

The Install SRK command authenticates and installs the root public key for use in subsequent Install CSFK (HAB only) or Install Key (HABv4 only) commands.

The HAB or AHAB authenticates the SRK using the SRK hash (`SRK_HASH`) eFuses. HABv4 or AHAB allows you to revoke individual keys within the SRK table, using the SRK revocation (`SRK_REVOKE`) eFuses.

HAB installs the SRK in slot 0 of its internal public key store.

Each CSF file must contain exactly one Install SRK command that appears before the Install CSFK (HAB only) command.

Table 8 lists the Install SRK command arguments.

All information provided in this document is subject to legal disclaimers.

© 2024 NXP B.V. All rights reserved.

Table 8. Install SRK arguments

Argument	Description	Valid values	HABv4	AHAB
File	SRK table or SRK table array (if the target is AHAB with hybrid container format)	Valid file path	M	M
Source Index	SRK index within SRK table. If the SRK revocation eFuse with this index is burned, installation fails.	0, 1, 2, and 3	M	M
Source	SRK certificate corresponding to the specified SRK index	Valid file path	X	M
Source Set	Origin of the SRK table	• NXP (it is reserved for NXP deliverables) • OEM	X	M
Revocations	Revoked SRKs ***Note:*** *This field can trigger a fusing procedure.*	4-bit bitmask	X	M
Hash Algorithm	SRK table hash algorithm	SHA-256	D	X

In Table 8,

- M = Mandatory
- D = Use default from Header if absent
- X = Not present

6.2.2.1 Install SRK usage

The following code snippet shows how to use the Install SRK command in a HABv4 example:

```
[Install SRK] # HAB4 example
    File = "../crts/srk_table.bin"
    Source Index = 0
    Hash Algorithm = sha256
```

The following code snippet shows how to use the Install SRK command in an AHAB example:

```
[Install SRK] # AHAB example
    File = "../crts/srk_table.bin"
    Source = "../crts/srk3_crt.pem"
    Source index = 2
    Source set = OEM
    Revocations = 0x0
```

6.2.3 Install CSFK (HAB only)

The Install CSFK command authenticates and installs a public key for use in subsequent Authenticate CSF commands.

The HAB authenticates the CSFK from the CSFK certificate using the SRK. HAB installs the CSFK in slot 1 of its internal public key store.

Each CSF file must contain exactly one Install CSFK command that appears before the Authenticate CSF command.

All information provided in this document is subject to legal disclaimers.

© 2024 NXP B.V. All rights reserved.

Table 9 lists the Install CSFK command arguments.

Table 9. Install CSFK arguments

Argument	Description	Valid values	HABv4
File	CSFK certificate	Valid file path	M
Certificate Format	CSFK certificate format	X509	D

In Table 9,

- M = Mandatory
- D = Use default from Header if absent

6.2.3.1 Install CSFK usage

The following code snippet shows how to use the Install CSFK command:

```
[Install CSFK] # HAB4 example
File = "../crts/csf.pem"
Certificate Format = X509
```

6.2.4 Install NOCAK (HABv4 only)

The Install NOCAK command authenticates and installs a public key for use with the fast authentication mechanism (HAB 4.1.2 and later only). With this mechanism, one key is used for all signatures.

HAB installs the no-CA key in slot 1 of its internal public key store.

Each CSF file must contain exactly one Install NOCAK command that appears before the Authenticate CSF command. Also, it must not contain any Install Key command.

Table 10 lists the install NOCAK command arguments.

Table 10. Install NOCAK arguments

Argument	Description	Valid values	HABv4
File	CSFK certificate	Valid file path	M
Certificate Format	CSFK certificate format	X509	D

In Table 10,

- M = Mandatory
- D = Use default from Header if absent

6.2.4.1 Install NOCAK usage

The following code snippet shows how to use the Install NOCAK command:

```
[Install NOCAK] # HAB4 example
File = "../crts/csf.pem"
Certificate Format = X509
```

6.2.5 Authenticate CSF (HAB only)

The Authenticate CSF command authenticates the CSF from which it is executed.

All information provided in this document is subject to legal disclaimers.

© 2024 NXP B.V. All rights reserved.

The HAB authenticates the CSF using the CSFK public key from a digital signature generated by the CST.

Each CSF file must contain exactly one Authenticate CSF command that appears after the Install CSFK command. Most other CSF commands are allowed only after the Authenticate CSF command.

Table 11 lists the Authenticate CSF command arguments.

Table 11. Authenticate CSF arguments

Argument	Description	Valid values	HABv4
Engine	CSF signature hash engine	ANY, SAHARA, RTIC, DCP, CAAM, and SW	D
Engine Configuration	Configuration flags for the hash engine. ***Note:*** *The hash is computed over an internal RAM copy of the CSF.*	See Table 7	D
Signature Format	CSF signature format	CMS	D

6.2.5.1 Authenticate CSF usage

The following code snippet shows how to use the Authenticate CSF command:

```
[Authenticate CSF] # HAB4 example using all default arguments
[Authenticate CSF] # HAB4 example
Engine = DCP
Engine Configuration = 0
Signature Format = CMS
```

6.2.6 Install Key (HABv4 only)

The Install Key command authenticates and installs a public key for use in subsequent Install Key or Authenticate Data commands.

The HAB authenticates a public key from a public key certificate using a previously installed verification key and a hash of the public key certificate.

HAB installs the authenticated public key in an internal public key store with a zero-based array of key slots.

The CSF author manages the key slots in the internal public key store by:

- Establishing the desired public key hierarchy
- Determining the keys used in authentication operations

Overwriting an occupied key slot is not allowed; however, reinstalling a public key at its own slot does not generate an error.

Multiple Install Key commands are allowed in a CSF. An Install Key command must precede any command that uses the installed key. Moreover, all Install Key commands must come after the Authenticate CSF command.

Table 12 lists the Install Key command arguments.

Table 12. Install Key arguments

Argument	Description	Valid values	HABv4
File	Public key certificate	Valid file path	M
Verification Index	Verification key index in key store	0, 2, 3, and 4. CSFK is not supported.	M

All information provided in this document is subject to legal disclaimers.

© 2024 NXP B.V. All rights reserved.

Table 12. Install Key arguments*...continued*

Argument	Description	Valid values	HABv4
Target Index	Target key index in key store	2, 3, and 4. SRK and CSFK slots are reserved.	M
Certificate Format	Public key certificate format	X509	D
Hash Algorithm	Hash algorithm for certificate binding. If the Hash Algorithm argument is present, the hash of the certificate specified in the File argument is included in the command. It prevents installation from other public key certificates sharing the verification key.	SHA-256	O

6.2.6.1 Install Key usage

The following code snippet shows how to use the Install Key command:

```
[Install Key] # HAB4 example
Key = "../crts/imgk.pem"
Verification Index = 0
Target Index = 2
Certificate Format = X509
```

6.2.7 Authenticate Data

The Authenticate Data command verifies the authenticity of the preloaded data in the memory. The data includes executable software instructions. It can be spread across multiple non-contiguous address ranges drawn from multiple object files.

The HAB authenticates the preloaded data using a previously installed public key from a digital signature generated by the CST.

Table 13 lists the Authenticate Data command arguments.

Table 13. Authenticate Data arguments

Argument	Description	Valid values	HABv4	AHAB
Blocks	A list of one or more data blocks, with each block having the following four parameters: • Source file (must be *binary*) • Starting load address in memory • Starting offset within the source file • Length (in bytes) ***Note:*** *A maximum of eight statement blocks are allowed.*	`file address offset length` with: • `file`: Valid path name • `address`: 32-bit unsigned integer • `offset`: 0, 1, 2, and so on (size of `file`) • `length`: 0, 1, 2, and so on (size of `file` - `offset`) Block parameters are separated by a whitespace. Multiple blocks are separated by commas.	M	X
Verification Index	Verification key index in the key store	2, 3, and 4 (HABv4). SRK and CSFK are not supported. ***Note:*** *For HABv4 fast authentication, the value of the Verification Index argument must be 0.*	M	X

All information provided in this document is subject to legal disclaimers.

© 2024 NXP B.V. All rights reserved.

Table 13. Authenticate Data arguments*...continued*

Argument	Description	Valid values	HABv4	AHAB
Engine	Data signature hash engine	ANY, SAHARA, RTIC, DCP, CAAM, and SW	D	X
Engine Configuration	Configuration flags for the engine	See Table 7	D	X
Signature Format	Data signature format	CMS	D	X
File	Binary to be signed	Valid file path	X	M
Offsets	A pair of offsets. The CST uses this information while signing the binary. The information can be retrieved using the `imx-mkimage` command.	`container_header_offset` and `signature_block_offset`. Offset parameters are separated by a whitespace. The values are unsigned integers.	X	M
Signature	A binary file containing the signature of the container. This field has been added for Hardware Security Module (HSM) support.	Valid file path	X	O

6.2.7.1 Authenticate Data usage

The following code snippet shows how to use the Authenticate Data command:

```
[Authenticate Data] # HAB4 example
Blocks = 0xf8000000 0x0 0x10000 "flash.bin", \
           0xf801000 0x0 0x1000 "xyz.bin"
Verification Index = 2
Engine = DCP
Engine Configuration = 0
Signature Format = CMS

[Authenticate Data] # AHAB example
File = "flash.bin"
Offsets = 0x400 0x610
```

6.2.8 Install Secret Key

This command is applicable from HAB 4.1 onward and only on processors that include CAAM and SNVS. Each instance of this command generates a CSF command to install a secret key in the secret key store of CAAM. As described in Section 2.1.2, a key blob is unwrapped using a master key encryption key (KEK) supplied by SNVS.

The CST back end generates and protects a random key. The key is encrypted using a public key passed to the CST with the command-line option `--cert`. It is saved in a file, which is named according to the Key argument. Later, the provisioning software uses this file to create the blob.

Table 14 lists the Install Secret Key command arguments. Each execution of the CST generates a different secret key, overwriting any secret key mentioned previously in the given file.

Table 14. Install Secret Key arguments

Argument	Description	Valid values	HABv4	AHAB
Key	Output filename for CST to create the encrypted data encryption key	Valid path name	M	M

All information provided in this document is subject to legal disclaimers.

© 2024 NXP B.V. All rights reserved.

Table 14. Install Secret Key arguments*...continued*

Argument	Description	Valid values	HABv4	AHAB
Key Length	Key length in bits	128, 192, and 256	M	M
Verification Index	Master KEK index	• 0 or 1: OTPMK from eFuses • 2: ZMK from SNVS • 3: CMK from SNVS	D	X
Target Index	Target secret key store index	0, 1, 2, or 3 of the secret key store	M	X
Blob Address	Absolute memory address where the blob is loaded	Internal or external DDR address	M	X
Key Identifier	Identifier that must match the value provided during the blob generation	32-bit value. The default value is 0.	X	O
Image Indexes	List of images to be encrypted	Mask of bits. By default, all images are encrypted.	X	O

6.2.8.1 Install Secret Key usage

The following code snippet shows how to use the Install Secret Key command:

```
[Install Secret Key] # HAB4 - Example using OTPMK (Default)
Key = "data_encryption.key"
Target Index = 0 /* Secret key store index */
Blob Address = 0x0090a000 /* internal ram address */
[Install Secret Key] # HAB4 - Example using ZMK
Key = "data_encryption.key"
Verification Index = 2 /* ZMK */
Target Index = 0 /* Secret key store index */
Blob Address = 0x0090a000 /* internal ram address */
[Install Secret Key] # AHAB - Example using default values
Key = "data_encryption.key"
Key Length = 128
[Install Secret Key] # AHAB - Example using optional values
Key = "data_encryption.key"
Key Length = 128
Key Identifier = 0x4a534d21
Image Indexes = 0xFFFFFFFE /* Image index 0 not encrypted */
```

6.2.9 Decrypt Data (HAB only)

This command is applicable from HABv4.1 onward. Each instance generates a CSF command to decrypt and authenticate a list of code / data blocks, using a secret key stored in the secret key store. The CST generates a corresponding `AUT_DAT` command. The CST encrypts the data blocks in place in the given files using a secret key. It also generates the medium access control (MAC) data, which is appended to the CSF.

Table 15 lists the Decrypt Data command arguments. The secret key index must be specified as the target key index in a preceding Install Secret Key command. The same secret key must not be used again. The Decrypt Data command removes the used secret key from the secret key store. A separate Install Secret Key command (which generates a fresh secret key) is required for another Decrypt Data command.

All information provided in this document is subject to legal disclaimers.

© 2024 NXP B.V. All rights reserved.

Table 15. Decrypt Data arguments

Argument	Description	Valid values	HAB (> 4.0)
Blocks	A list of one or more data blocks, with each block having the following four parameters: • Source file (must be *binary*) • Starting load address in memory • Starting offset within the source file • Length (in bytes)	`file address offset length` with: • `file`: Valid path name • `address`: 32-bit unsigned integer • `offset`: 0, 1, 2, and so on (size of `file`) • `length`: 0, 1, 2, and so on (size of `file` - `offset`) Block parameters are separated by a whitespace. Multiple blocks are separated by commas.	M
Verification Index	Secret key index in the secret key store	0, 1, 2, and 3 from secret key store	M
Engine	MAC engine	CAAM (default value)	D
Engine Configuration	Configuration flags for the engine	See Table 7 The default value is from the Header command.	D
MAC Bytes	Size of MAC in bytes	An even value in the range from 4 to 16. The default value is 16.	D

6.2.9.1 Decrypt Data usage

The following code snippet shows how to use the Decrypt Data command:

```
[Decrypt Data]
Blocks = 0xf8000000 0x0 0x10000 "flash.bin", \
0xf8010000 0x0 0x1000 "xyz.bin"
Verification Index = 0

[Decrypt Data]
Blocks = 0xf8000000 0x0 0x10000 "flash.bin", \
          0xf8010000 0x0 0x1000 "xyz.bin", \
          0xf8012000 0x2000 0x4000 "xyz.bin", \
          0xf8018000 0x8000 0x1000 "xyz.bin"
Verification Index = 3
Engine = CAAM
Engine Configuration = 0
```

6.2.10 NOP (HAB only)

The NOP command has no effect.

In a CSF, multiple NOP commands can appear after the Authenticate CSF command. For HABv4, NOP commands can also appear between the Header and Authenticate CSF commands.

The NOP command has no argument.

6.2.10.1 NOP usage

The NOP command can be used as follows:

```
[NOP]
```


All information provided in this document is subject to legal disclaimers.

© 2024 NXP B.V. All rights reserved.

6.2.11 Set Engine (HAB only)

The Set Engine command selects the default engine and engine configuration for a given algorithm.

Some CSF commands allow the CSF author to select the engine used for an algorithm by specifying an argument other than ANY. However, if the engine argument is ANY, the HAB selects the required engine based on an internal criteria. The Set Engine command overrides the HAB internal criteria and selects the engine and configuration to use, when ANY is specified.

Some algorithm types do not have an associated engine argument in the CSF commands, for example, the signature algorithm in the Authenticate Data commands. By default, the HAB selects the engine to use for such algorithms based on an internal criteria. The Set Engine command also overrides the HAB internal criteria in such scenarios.

In a CSF, multiple Set Engine commands can appear after the Header command. A subsequent Set Engine command uses the engine selected by the previous Set Engine command.

Table 16 lists the Set Engine command arguments.

Table 16. Set Engine arguments

Argument	Description	Valid values	HABv4
Hash Algorithm	Hash algorithm	SHA-256	M
Engine	Engine. Use ANY to restore the HAB internal criteria.	ANY, SAHARA, RTIC, DCP, CAAM, and SW	M
Engine Configuration	Configuration flags for the engine	See Table 7	O

6.2.11.1 Set Engine usage

The following code snippet shows how to use the Set Engine command:

```
[Set Engine]
Hash Algorithm = SHA256
Engine = DCP
Engine Configuration = 0
```

6.2.12 Init (HAB only)

The Init command initializes the specified engine features during exit from the internal boot ROM.

Multiple Init commands can appear after the Authenticate CSF command. If one or more Init commands specify a feature, the feature is initialized.

Table 17 lists the Init command arguments.

Table 17. Init arguments

Argument	Description	Valid values	HABv4
Engine	Engine to initialize	SRTC	M
Features	Comma-separated list of features to initialize	RNG (CAAM) (see Table 19)	O

Note: *Before using the Init RNG feature, refer to i.MX Secure Boot on HABv4 Supported Devices (AN4581) (see Table 25 for more details).*

All information provided in this document is subject to legal disclaimers.

© 2024 NXP B.V. All rights reserved.

6.2.12.1 Init usage

The following code snippet shows how to use the Init command:

```
[Init]
Engine = SRTC
[Init]
Engine = CAAM
Features = RNG
```

6.2.13 Unlock (HAB only)

The Unlock command prevents the specified engine features from being locked during exit from the internal boot ROM.

Multiple Unlock commands can appear after the Authenticate CSF command. If one or more Unlock commands specify a feature, the feature is unlocked.

Table 18 lists the Unlock command arguments.

Table 18. Unlock arguments

Argument	Description	Valid values	HABv4
Engine	Engine to unlock	SRTC, CAAM, SNVS, and OCOTP	M
Features	Comma-separated list of features to unlock	See Table 19	O
UID	Device specific 64-bit UID. It is required to unlock certain features, and it must be absent for other features (see Table 19).	U0, U1, U2, U3, U4, U5, U6, and U7 with Ui = 0 – 255. UID bytes are separated by commas.	M/X

In Table 18,

- M = Mandatory
- O = Optional
- X = Not present

Table 19 shows the valid Features values available in the Init/Unlock command for each Engine argument.

Table 19. Valid Features values

Engine	Features	UID	Init/Unlock command effect
SRTC		X	If the SRTC is in Init state, the Init command clears any failure status flags and clears the low-power counters and timers. If the SRTC is in Valid state, the Unlock command prevents the secure timer and the monotonic counter from being locked.
CAAM	MID	X	The Job Ring and DECO Master ID registers are left unlocked.
	RNG	X	• The RNG state handle 0 is left uninstantiated. • The descriptor keys are not generated. • The AES DPA mask is not set. • The state handle 0 test instantiation is not blocked.
	MFG	X	Keep manufacturing protection private key in the CAAM internal memory.
SNVS	LP SWR	X	The low-power (LP) software reset is left unlocked.
	ZMK WRITE	X	The zeroizable master key (ZMK) write is left unlocked.

All information provided in this document is subject to legal disclaimers.

© 2024 NXP B.V. All rights reserved.

Table 19. Valid Features values*...continued*

Engine	Features	UID	Init/Unlock command effect
OCOTP	FIELD RETURN	M	The field return activation is left unlocked.
	SRK REVOKE	X	The SRK revocation is left unlocked.
	SCS	M	The SCS register is left unlocked.
	JTAG	M	Unlock JTAG using the SCS register bit HAB_JDE.

In Table 19,

- M = Mandatory
- X = Not present

6.2.13.1 Unlock usage

The following code snippet shows how to use the Unlock command:

```
[Unlock]
    Engine = SRTC
[Unlock]
    Engine = CAAM
    Features = RNG
[Unlock]
    Engine = OCOTP
    Features = JTAG, SRK REVOKE
    UID = 0x01, 0x23, 0x45, 0x67, 0x89, 0xab, 0xcd, 0xef
```

6.2.14 Install Certificate (AHAB only)

The Install Certificate command is optional. It converts a public key into the NXP format.

The AHAB authenticates a certificate using a previously installed verifying SRK and a hash of the public key certificate.

Each CSF file must contain up to one Install Certificate command. Table 20 lists the Install Certificate command arguments.

Table 20. Install Certificate arguments

Argument	Description	Valid values	AHAB
File	Public key certificate	Valid file path	M
Permissions	Refer to the AHAB architecture specification in the device security reference manual for setting the value of this argument correctly.	8-bit bitmask	M
Signature	A binary file containing the signature of the NXP-format public key certificate. This field has been added for Hardware Security Module (HSM) support.	Valid file path	O
For AHAB version 2 only			
Permissions Data	Complementary information for debug authentication feature	3 words of 32-bit each	O
UUID	Unique ID of the target device	4 words of 32-bit each	O
Fuse Version	Version of certificate	8-bit value	O

All information provided in this document is subject to legal disclaimers.

© 2024 NXP B.V. All rights reserved.

6.2.14.1 Install Certificate usage

The following code snippet shows how to use the Install Certificate command:

```
[Install Certificate]
File = "../crts/sgk1_crt.pem"
Permissions = 0x1
```

6.3 CSF examples

This section provides some examples of HABv4 and AHAB CSF files.

6.3.1 HABv4 CSF example

Code listing "Example HABv4 CSF description file" shows an example HABv4 CSF description, which:

- Defines a version 4 CSF description
- Overrides the default value (ANY) for the Engine argument in the Authenticate Data command with DCP
- Lists three blocks from the image for signing

Code listing: Example HABv4 CSF description file

```
[Header]
    Version = 4.0
    Security Configuration = Open
    Hash Algorithm = sha256
    Engine = DCP
        Engine Configuration = 0
        Certificate Format = X509
        Signature Format = CMS

[Install SRK]
        File = "TBL_1_sha256_tbl.bin"
        Source index = 0

[Install CSFK]
        File = "CSF1_1_pkcs1_pkcs1_sha256_1024_3_v3_usr_crt.bin"

[Authenticate CSF]

[Install Key]
        Verification index = 0
        Target index = 2
        File = "IMG1_1_pkcs1_pkcs1_sha256_1024_3_v3_usr_crt.bin"

# whole line comment

[Authenticate Data]   # part line comment
        Verification index = 2
        Engine = DCP
        Blocks = 0xf8009400 0x400 0x40 "MCUROM-OCRAM-ENG_img.bin", \
                 0xf8009440 0x440 0x40 "MCUROM-OCRAM-ENG_img.bin", \
                 0xf800a000 0x1000 0x8000 "MCUROM-OCRAM-ENG_img.bin"
```


All information provided in this document is subject to legal disclaimers.

© 2024 NXP B.V. All rights reserved.

6.3.2 HABv4 CSF fast authentication example

Code listing "Example HABv4 CSF description file for fast authentication" shows an example HABv4 CSF description for fast authentication. This example CSF description:

- Defines a version 4 CSF description
- Instructs the HAB to use the fast authentication mechanism
- Lists a single block from the image for signing

Code listing: Example HABv4 CSF description file for fast authentication

```
#Illustrative Command Sequence File Description
[Header]
    Version = 4.1
    Hash Algorithm = sha256
    Engine = ANY
    Engine Configuration = 0
    Certificate Format = X509
    Signature Format = CMS

[Install SRK]
    File = "../crts/TBL_1_sha256+tbl.bin"
    Source index = 0

[Install NOCAK]
    File = "../crts/SRK1_sha256_2048_65537_v3_usr_crt.pem"

[Authenticate CSF]
#whole line comment

[Authenticate Data]    # part line comment
    Verification index = 0
    Blocks = 0x877fb000 0x000 0x48000 "signed-uboot.bin"
```

6.3.3 HABv4 CSF example for encrypted boot

Code listing "Example HABv4 CSF description file with Decrypt Data command" shows an example HABv4 CSF description for encrypted boot. This example CSF description:

- Defines a version 4.1 CSF description
- Explains how to use the Install Secret Key and Decrypt Data commands
- Lists necessary blocks from image for signing
- Lists blocks that CST uses for encryption and blocks that ROM/HAB uses for decryption

Code listing: Example HABv4 CSF description file with Decrypt Data command

```
[Header]
    Version = 4.1
    Hash Algorithm = SHA256
    Engine Configuration = 0
    Certificate Format = X509
    Signature Format = CMS
    Engine = CAAM
    Engine Configuration = 0

[Install SRK]
    File = "../crts/SRK_1_2_3_4_table.bin"
    Source index = 0
```


All information provided in this document is subject to legal disclaimers.

© 2024 NXP B.V. All rights reserved.

```
[Install CSFK]
    File = "../crts/CSF1_1_sha256_4096_65537_v3_usr_crt.der"

[Authenticate CSF]

[Install Key]
    Verification index = 0
    Target index = 2
    File = "../crts/IMG1_1_sha256_4096_65537_v3_usr_crt.der"

[Authenticate data]
    Verification index = 2
    Blocks = 0x27800400 0x400 800 "u-boot-mx6q-arm2_padded.bin"

[Install Secret Key]
    Verification index = 0
    Target index = 0
    Key = "dek.bin"
    Key Length = 128
    Blob address = 0x27831000

[Decrypt Data]
    Verification index = 0
    Mac Bytes = 16
    Blocks = 0x27800720 0x720 0x2E8E0 "u-boot-mx6q-arm2_padded.bin"
```

6.3.4 AHAB CSF example

Code listing "Example AHAB CSF description file" shows an example AHAB CSF description.

Code listing: Example AHAB CSF description file

```
[Header]
    Target = AHAB
    Version = 1.0

[Install SRK]
    # Output of srktool
    File = "…/crts/srk_table.bin"
    # Public key certificate in PEM or DER format
    Source = "…/crts/srk1_crt.pem"
    # Index of SRK in SRK table
    Source index = 0
    # Origin of SRK table
    Source set = OEM
    # Revoked SRKs
    Revocations = 0x0

[Authenticate Data]
    # Output of mkimage
    File = "flash.bin"
    # Offsets = Container header Signature block (printed out by mkimage)
    Offsets   = 0x400             0x490
```


All information provided in this document is subject to legal disclaimers.

© 2024 NXP B.V. All rights reserved.

6.3.5 AHAB CSF with certificate example

Code listing "Example AHAB CSF description file with certificates" shows an example AHAB CSF description with certificates.

Code listing: Example AHAB CSF description file with certificates

```
[Header]
    Target = AHAB
    Version = 1.0

[Install SRK]
    # Output of srktool
    File = "…/crts/srk_table.bin"
    # Public key certificate in PEM or DER format
    Source = "…/crts/srk3_crt.pem"
    # Index of SRK in SRK table
    Source index = 2
    # Origin of SRK table
    Source set = OEM
    # Revoked SRKs
    Revocations = 0x1
[Install Certificate]
    # Public key certificate in PEM or DER format
    File = "…/crts/sgk3_crt.pem"
    Permissions = 0x1

[Authenticate Data]
    # Output of mkimage
    File = "flash.bin"
    # Offsets = Container header Signature block (printed out by mkimage)
    Offsets   = 0x400             0x710
```

6.3.6 AHAB CSF example for encrypted boot

Code listing "Example AHAB CSF description file for encrypted boot" shows an example AHAB CSF description for encrypted boot.

Code listing: Example AHAB CSF description file for encrypted boot

```
[Header]
    Target = AHAB
    Version = 1.0

[Install SRK]
    # Output of srktool
    File = "…/crts/srk_table.bin"
    # Public key certificate in PEM or DER format
    Source = "…/crts/srk1_crt.pem"
    # Index of SRK in SRK table
    Source index = 0
    # Origin of SRK table
    Source set = OEM
    # Revoked SRKs
    Revocations = 0x0

[Authenticate Data]
    # Output of mkimage
    File = "flash.bin"
```


All information provided in this document is subject to legal disclaimers.

© 2024 NXP B.V. All rights reserved.

```
    # Offsets = Container header Signature block (printed out by mkimage)
    Offsets   = 0x400             0x490

[Install Secret Key]
    Key = "data_encryption.key"
    Key Length = 128
    Key Identifier = 0x4a534d21
    # Image index 0 not encrypted
    Image Indexes = 0xFFFFFFFE
```

6.3.7 AHAB CSF example for hybrid container signing

Code listing "Example AHAB CSF description file for hybrid container signing" shows an example AHAB CSF description for signing a hybrid container.

Code listing: Example AHAB CSF description file for hybrid container signing

```
[Header]
Target = AHAB Version = 2.0

[Install SRK]
# Output of srktool (SRK table array)
File = "…/crts/srk_tables.bin"
# Public key certificate in PEM or DER format Source = "…/crts/srk1_crt.pem"
# Index of SRK in SRK table Source index = 0
# Origin of SRK table Source set = OEM
# Revoked SRKs Revocations = 0x0

[Authenticate Data]
# Output of mkimage File = "flash.bin"
# Offsets = Container header Signature block (printed out by mkimage)
Offsets = 0x400 0x490
```


All information provided in this document is subject to legal disclaimers.

© 2024 NXP B.V. All rights reserved.

7 CST architecture

The NXP CST is a reference implementation and is sufficient for most use cases. The tool has a front end that supports the NXP proprietary operations. It also has two reference back-end implementations, which provide cryptographic services to the front end through the adaption layer. Figure 12 provides an overview of reference CST components.

The first back end performs all cryptographic operations related to digital signature generation and encryption, using the OpenSSL library (see Section 8). It accesses the key material directly from the filesystem.

The second back end performs cryptographic operations using an OpenSSL engine that supports the PKCS#11 interface. It references the key material through token identifiers in the CSF description files. For information on how to use the PKC#11 back end with a provider, see Section 7.3.

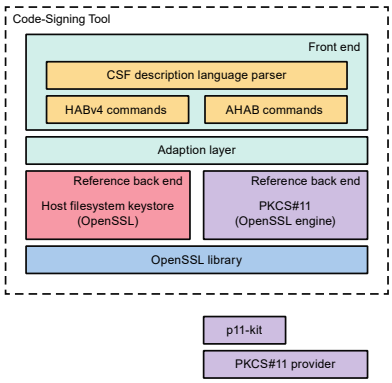

Figure 12. Overview of reference CST components

Library dependencies:

CST version 4.0.0 is built with the following libraries:

- OpenSSL version 3.2.0: It is used for cryptographic operations, including signature verification and certificate management.
- libp11 version 0.4.12: It is needed for interfacing with the PKCS#11 modules.
- liboqs version 0.10.1: It is needed for post-quantum cryptographic operations.
- oqsprovider version 0.6.1: It is required as an OpenSSL provider to support hybrid and PQC algorithms.
- json-c version 0.17: It is needed for JSON handling in the ahab_signed_message tool.
- libhidapi-libusb.so version 0.14.0: It is required to allow the hab_log_parser tool to interface with the i.MX device over USB.

7.1 Customizing CST back end

To provide additional cryptographic services, you may require to modify the CST reference implementation. To meet this requirement, the CST is designed in two parts:

- Front end: Contains all NXP proprietary operations of the CST
- Back end: Contains all standard cryptographic operations

In addition to the CST executables, the package includes the source code, which is located in the `code/cst` directory of the package. The package also includes a Dockerfile. You can use the Dockerfile to create a build environment or as a reference to determine the dependency requirements of the build host.

Although you can replace OpenSSL in the CST back end with a new tool to perform code signing and SA and CA functions. However, OpenSSL is still needed in the CST front end for some non-code signing operations. It

All information provided in this document is subject to legal disclaimers.

© 2024 NXP B.V. All rights reserved.

means that when linking library components together to generate a CST executable, an OpenSSL library must also be included. CST release 3.4.0 and later use OpenSSL 3.2.0, which is available at http://www.openssl.org.

7.1.1 Back-end APIs

The back end must implement the following APIs that are used by the front end:

- `gen_sig_data()`
- `gen_auth_encrypted_data()`
- `read_certificate()`

Any new back-end implementation must implement these APIs in an equivalent adaptation layer corresponding to the new cryptographic services. The source code and header files of the NXP reference implementation are provided for reference purposes. To use a new method for public key generation, create/use new key generation scripts.

7.1.1.1 `gen_sig_data()`

The CST front end uses the `gen_sig_data()` API to generate signature data. The API is defined as follows:

```
int32_t gen_sig_data(const char* in_file,
                     const char* cert_file,
                     hash_alg_t hash_alg,
                     sig_fmt_t sig_fmt,
                     uint8_t* sig_buf,
                     size_t *sig_buf_bytes,
                     func_mode_t mode);
```

Table 21 explains the `gen_sig_data()` parameters.

Table 21. `gen_sig_data()` parameters

Parameter	Description
`in_file`	Input data
`cert_file`	Signer certificate file
`hash_alg`	Hash algorithm
`sig_fmt`	Signature format
`sig_buf`	Signature buffer
`sig_buf_bytes`	Size of signature buffer
`Mode`	Custom mode

7.1.1.2 `gen_auth_encrypted_data()`

The CST front end uses the `gen_auth_encrypted_data()` API to generate authenticated encrypted data. The API generates an encryption key and uses it to encrypt plaintext input data. The API can optionally use an existing input key, instead of generating a new key. A generated encryption key can optionally be encrypted using a certificate for secure transport.

The `gen_auth_encrypted_data()` API is defined as follows:

```
int32_t gen_auth_encrypted_data(const char* in_file,
                                const char* out_file,
                                aead_alg_t aead_alg,
                                uint8_t *aad,
```


All information provided in this document is subject to legal disclaimers.

© 2024 NXP B.V. All rights reserved.

```
                                    size_t aad_bytes,
                                    uint8_t *nonce,
                                    size_t nonce_bytes,
                                    uint8_t *mac,
                                    size_t mac_bytes,
                                    size_t key_bytes,
                                    const char* cert_file,
                                    const char* key_file,
                                    int reuse_dek);
```

Table 22 explains the `gen_auth_encrypted_data()` parameters.

Table 22. `gen_auth_encrypted_data()` parameters

Parameter	Description
`in_file`	Plaintext input data
`out_file`	Output cipher text
`aead_alg`	AES_CCM or AES_CCB
`Aad`	Additional authenticated data (AAD)
`aad_bytes`	Size of AAD
`Nonce`	Nonce bytes to return
`nonce_bytes`	Size of nonce
`Mac`	Output MAC
`mac_bytes`	Size of MAC
`key_bytes`	Size of symmetric key
`cert_file`	Certificate file for DEK encryption
`key_file`	Input key file
`reuse_dek`	Use existing input key

7.1.1.3 `read_certificate()`

The `read_certificate()` API reads X.509 certificate data from the given certificate file. The API is defined as follows:

```
X509* read_certificate(const char* reference);
```

Table 23 explains the `read_certificate()` parameter.

Table 23. `read_certificate()` parameter

Parameter	Description
`reference`	Certificate file

7.2 Front-end references to code signing keys

When replacing the CST back end, remember that the CST front end refers to code signing keys and certificates using key filenames. These filenames correspond to the RSA public key certificate and private key files generated by the CST PKI scripts. In a replacement back-end service, filenames may not be the native method for referencing keys. In such a scenario, the new adaptation layer is responsible for converting to and from filename references. It is also true for data encryption keys that the CST generates for encrypting images.

All information provided in this document is subject to legal disclaimers.

© 2024 NXP B.V. All rights reserved.

7.3 Using CST with Hardware Security Module

This section explains how to use the CST effectively with a Hardware Security Module (HSM).

The default back end of the CST uses OpenSSL to perform HAB and AHAB signature generation and data encryption. OpenSSL exposes an engine API, which allows you to add alternative implementations for its native cryptographic operations.

Libp11 (openssl-pkcs11) serves as the PKCS#11 engine that the CST uses to access PKCS#11 enabled HSMs. It contains the functions required for session and token management, certificate handling, signing, and hashing. With these functions, Libp11 acts as an interface between vendor PKCS#11 modules and the OpenSSL engine API. This engine can communicate directly with a PKCS#11 provider or use the p11-kit proxy module for system-wide PKCS#11 module access. For more details, visit p11-kit web pages.

The steps to use the CST with the HSM are provided below.

Step 1: Build and run the `Dockerfile.hsm` image

The root folder of the package contains a `Dockerfile.hsm` file, which allows you to install dependencies in a seamless manner. It enables you to create a setup quickly for working with the CST, along with the PKCS#11 back end. Build the `Dockerfile.hsm` image as follows:

```
$ docker build -t cst:hsm --build-arg http_proxy=$http_proxy --build-arg
 hostUserName=$USER --build-arg hostUID=$(id -u) --build-arg hostGID=$(id -g) -f
 Dockerfile.hsm
```

Launch a shell that can be used to run the commands as user *SoftHSM* for the subsequent steps:

```
$ docker run -v $(pwd):/home/$USER/cst -w /home/$USER/cst --rm -it --entrypoint
 bash cst:hsm
```

Step 2: Create a token

Typically, the `pkcs11-tool` command is used for interacting with the PKCS#11 tokens. To initialize a token, run the `pkcs11-tool` command as follows:

```
$ pkcs11-tool --module $PKCS11_MODULE_PATH --init-token --init-pin --so-pin=
$SO_PIN --new-pin=$USR_PIN --label="CST-HSM-DEMO" --pin=$USR_PIN --login

Using slot 0 with a present token (0x0)
Token successfully initialized
User PIN successfully initialized
```

`PKCS11_MODULE_PATH` indicates the name (including path) of the PKCS#11 module. In the current example, it is set to the path of the p11-kit proxy module. Based on the desired PKCS#11 configuration, `PKCS11_MODULE_PATH` can be changed to the path of the SoftHSM. Later, a label (identifier) in the *CST-HSM-DEMO* token can be used with the CST to locate the certificates and keys needed to sign the images.

Step 3: Generate a PKI tree

To generate a PKI tree for AHAB or HABv4, run the `hsm_ahab_pki_tree.sh` or `hsm_hab4_pki_tree.sh` script, respectively, in Interactive or CLI mode. The script prompts you with a series of questions that you have to answer to guide the key generation process.

In CLI mode, the corresponding commands are as follows:

```
$ cd keys/
$ ./hsm_hab4_pki_tree.sh
```


All information provided in this document is subject to legal disclaimers.

© 2024 NXP B.V. All rights reserved.

or

```
$ cd keys/
$ ./hsm_ahab_pki_tree.sh
```

The following is an example with arguments:

```
$./hsm_hab4_pki_tree.sh -existing-ca n -use-ecc n -kl 2048 -duration 10 -num-srk
 4 -srk-ca y
```

To list the objects stored on the token, run the following command:

```
$ pkcs11-tool --module $PKCS11_MODULE_PATH -l --pin $USR_PIN --list-objects
```

With the above command, you should be able to find both the keys and certificates within the token.

Step 4: Generate an SRK table and eFuse

Similar to the default CST back end, create the SRK table and eFuse files for HABv4 or AHAB by specifying paths to the SRK certificate:

```
$ cd ../crts/
$ ../linux64/bin/srktool ...
```

For example:

```
$ ../linux64/bin/srktool -h 4 -t SRK_1_2_3_4_table.bin -e
 SRK_1_2_3_4_fuse.bin -d sha256 -c ./SRK1_sha256_2048_65537_v3_ca_crt.pem,./
SRK2_sha256_2048_65537_v3_ca_crt.pem,./SRK3_sha256_2048_65537_v3_ca_crt.pem,./
SRK4_sha256_2048_65537_v3_ca_crt.pem -f 1
```

Step 5: Create a Command Sequence File

CST with PKCS#11 back end finds keys and certificates in an HSM using PKCS#11 URIs. The URI format is defined in the "PKCS #11 URI Scheme" specification, which is described in RFC 7512 (see https://tools.ietf.org/html/rfc7512 for more details).

In this example, the CSF instructs the CST to access the certificate with the identifier or label *CSF1_1_sha256_2048_65537_v3_usr* in the PKCS#11 token/module, named as *CST-HSM-DEMO*. It provides the PIN value *12345678* as part of the authentication process.

An example command is given below:

```
[Install CSFK]
   File = "pkcs11:token=CST-HSM-
DEMO;object=CSF1_1_sha256_2048_65537_v3_usr;type=cert;pin-value=${USR_PIN}"
```

where:

- `token=CST-HSM-DEMO`: Specifies the PKCS#11 module or token being used. In this case, the token is named as `CST-HSM-DEMO`.
- `object=CSF1_1_sha256_2048_65537_v3_usr`: Indicates the specific object within the token. In this example, it refers to an object with the identifier or label `CSF1_1_sha256_2048_65537_v3_usr`.
- `type=cert`: Indicates that the object is a certificate.
- `pin-value=${USR_PIN}`: It is an optional parameter that represents the PIN needed to access the specified object.

Step 6: Sign using the PKCS#11 back end

All information provided in this document is subject to legal disclaimers.

© 2024 NXP B.V. All rights reserved.

The final step is to invoke the CST with the `-b` switch, specifying the PKCS#11 back end:

```
$ ../linux64/bin/cst -b pkcs11 -i example.csf -o signed-example.bin
CSF Processed successfully and signed data available in signed-example.bin
```


All information provided in this document is subject to legal disclaimers.

© 2024 NXP B.V. All rights reserved.

8 References

Table 24 lists the resources that were referenced to produce the current document.

Table 24. Reference resources

Resource	Link
Open Secure Socket Layer (OpenSSL)	http://www.openssl.org
RFC 3610: Counter with CBC-MAC (CCM)	http://www.ietf.org/rfc/rfc3610.txt
RFC 3852: Cryptographic Message Syntax (CMS)	http://www.ietf.org/rfc/rfc3852.txt
RFC 5280: Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile	http://www.ietf.org/rfc/rfc5280.txt
RSA Private-Key Cryptography Standard #8 (PKCS #8) - Private-Key Information Syntax Standard, version 1.2, RSA Laboratories	http://www.rsa.com/rsalabs
WAP Certificate and CRL Profiles (WAP-211-WAPCert), 22-May-2001	http://www.openmobilealliance.org
PKCS#11 wrapper library, 12/2023	https://github.com/OpenSC/libp11
The p11-kit web pages, 12/2023	http://p11-glue.freedesktop.org/p11-kit.html
The PKCS #11 URI Scheme, 12/2023	https://tools.ietf.org/html/rfc7512
Open Quantum Safe provider for OpenSSL (3.x)	https://github.com/open-quantum-safe/oqs-provider

The documents listed in Table 25 provide additional information on secure boot with NXP processors.

Table 25. Additional documents

Document	Link / how to obtain
High Assurance Boot Version 4 API Reference Manual (RM00298)	Included as part of the NXP reference CST release
Secure Boot on i.MX25, i.MX35, and i.MX51 using HABv3	AN4547.pdf
Secure Boot with i.MX28 HAB v4	AN4555.pdf
i.MX Secure Boot on HABv4 Supported Devices	AN4581.pdf
Secure Boot on AHAB Supported Devices	AN12312.pdf
i.MX Linux User's Guide	i.MX Linux User's Guide
i.MX 6 DQ/DQP/SDL/SX/UL/ULL/ULZ Security Reference Manual	nxp.com
i.MX 7 SD/ULP Security Reference Manual	nxp.com
i.MX 8 MQ/MM/MN/MP/ULP/QXP/DXP/QM/DM/DXL Security Reference Manual	nxp.com
Using i.MX 8 Security Controller Signed Messages	AN13770.pdf
i.MX 8ULP Processor Reference Manual	IMX8ULPRM.pdf
i.MX 9x Security Reference Manual	nxp.com
i.MX 93 Applications Processor Reference Manual	IMX93RM.pdf

All information provided in this document is subject to legal disclaimers.

© 2024 NXP B.V. All rights reserved.

9 Acronyms

Table 26 lists the acronyms used in this document.

Table 26. Acronyms

Acronym	Description
AES	Advanced Encryption Standard
AHAB	Advanced High Assurance Boot
AP	Application processor
API	Application programming interface
ASN.1	Abstract Syntax Notation One
CA	Certificate authority
CAAM	Cryptographic Acceleration and Assurance Module
CCM	Counter with CBC-MAC
CI/CD	Continuous integration and continuous delivery
CLI	Command-line interface
CMS	Cryptographic message syntax
CSF	Command Sequence File
CST	Code signing tool
DCD	Device controller driver
DDR	Double Data Rate
DEK	Data encryption key
DER	ASN.1 Distinguished Encoding Rules
EC	Elliptic curve
ECC	Elliptic curve cryptography
ECDSA	Elliptic Curve Digital Signature Algorithm
ELE	EdgeLock Secure Enclave
FW	Firmware
HAB	High Assurance Boot
HABv4	High Assurance Boot version 4
HSM	Hardware Security Module
IPL	Interrupt priority level
JSON	JavaScript Object Notation
MAC	Medium access control
MGF	Mask generation function
MMU	Memory Management Unit
OEM	Original equipment manufacturer
OQS	Open Quantum Safe
OS	Operating system

All information provided in this document is subject to legal disclaimers.

© 2024 NXP B.V. All rights reserved.

Table 26. Acronyms...*continued*

Acronym	Description
PEM	Privacy Enhanced Mail
PKCS	Public-Key Cryptography Standards
PKI	Public key infrastructure
PQC	Post-quantum cryptography
PSS	Probabilistic Signature Scheme (cryptographic signature scheme) by Mihir Bellare and Phillip Rogaway
QSPI	Quad Serial Peripheral Interface
RNG	Random Number Generator
ROM	Read-only memory
RSA	Public key encryption algorithm created by Rivest, Shamir, and Adleman
SA	Signature authority
SAHARA	Symmetric/Asymmetric Hash and Random Accelerator
SDP	Serial download protocol
SECO	Security Controller
SHA	Secure Hash Algorithm
SHE	Secure Hardware Extension
SoC	System-on-chip
SPL	Secondary Program Loader
SRK	Super root key
SW	Software
UID	Unique ID — A field in the processor and CSF identifying a device or group of devices
V2X	Vehicle-to-everything
XIP	Execute-in-place
XOF	eXtendable output function

All information provided in this document is subject to legal disclaimers.

© 2024 NXP B.V. All rights reserved.

10 Note about the source code in the document

Example code shown in this document has the following copyright and BSD-3-Clause license:

Copyright 2024 NXP Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. Neither the name of the copyright holder nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

All information provided in this document is subject to legal disclaimers.

© 2024 NXP B.V. All rights reserved.

11 Software release history

Table 27 summarizes the changes made to the code signing tool (CST) during different releases of the tool.

Table 27. Software release history

Release number	Release date	Change description
4.0.0	29 November 2024	• **New feature support**: – Added support of AHAB hybrid container format – Added support of post-quantum cryptography (PQC) and hybrid algorithms: – Enabled support for ML-DSA, Dilithium 3, and hybrid keys in the SRK tool and CST, with functionality for generating and managing PQC and hybrid PKI trees, SRK tables, SRK fuse values, and hybrid container signing – Added support for SHA-3 and SHAKE hash algorithms, including SHA3-256, SHA3-384, SHA3-512, SHAKE128 output 256, and SHAKE256 output 512 – Improved PKCS#11 support by allowing static compilation of the OpenSSL PKCS#11 engine with CST – Added support of NXP proprietary certificate format version 2 fields in CSF for AHAB debug authentication – Added a CSF miscellaneous command "Signature Size" to manage CMS signature buffer size in HSM mode, accommodating larger CMS structures – Switched to CMake as the primary build system and restructured the source tree – Introduced the `build.sh` script to simplify the build process and manage dependencies – Upgraded *Dockerfile* and *Dockerfile.hsm* to Ubuntu 22.04 to support OpenSSL 3 – Added compatibility for cross-compiling for Windows 64-bit environments using MinGW64, enabling support for 64-bit PKCS#11 modules – Introduced new tools for AHAB and HABv4: – *ahab_pki_tree*: A new tool that is designed to mimic the functionality of the original `ahab_pki_tree.sh` and `ahab_pki_tree.bat` scripts while adding support for PQC and hybrid key management – *ahab_signed_message*: Added to assist the users in generating signed messages for SECO devices – *ahab_split_container*: Splits NXP-signed AHAB containers into ELE and V2X containers, to support OEM double signing/ authentication feature – *ahab_image_verifier*: Parses and verifies AHAB containers – *hab4_mac_dump*: Formerly known as *mac_dump*, this tool dumps the location and size of MAC data from a HABv4 CSF binary data file – *hab4_image_verifier*: Parses HABv4 images

All information provided in this document is subject to legal disclaimers.

© 2024 NXP B.V. All rights reserved.

Table 27. Software release history...*continued*

Release number	Release date	Change description
		– *hab4_pki_tree*: Mimics the original `hab4_pki_tree.sh` and `hab4_pki_tree.bat` scripts, providing basic HABv4 PKI generation • **Quality improvements**: – Implemented a CI/CD pipeline with automated testing in a board farm for continuous integration and validation • **Bug fixes**: – Resolved issues related to signing in HSM mode, specifically addressing RSA, RSA-PSS, and ECDSA signature verification – Fixed an issue occurring with serial number calculations in X.509 certificates when the serial number is negative – Resolved an issue in HSM mode where identical tags were generated for CSF and IMG due to a seeding issue with the random number generator • **Documentation**: – Updated usage instructions across various tools – Reformatted documentation into the official NXP template • **Miscellaneous fixes**: – Minor code refinements – Updated warning messages – Introduced a code formatting configuration to enforce consistent styling
3.4.1	31 May 2024	Added HABv4 MAC dump tool as a workaround for the HABv4 encrypted boot failure issue
3.4.0	31 December 2023	• Transitioned CST to OpenSSL 3 following OpenSSL 1.1.1 end-of-life in September 2023 (no more public security fixes for OpenSSL 1.1.1 in CST after this date). CST has been compiled with OpenSSL 3.2.0 for enhanced security and features. • Simplified the PKCS#11 back end, eliminating redundant code • Introduced `Dockerfile.hsm` for experimenting with the PKCS#11 back end • PKCS#11 generation scripts are now located in the `keys` folder. • Removed repetitive PIN requests in the PKI generation scripts • The back end supports RSA-PSS. • Improved build system for efficiency and ease of use • Updated Dockerfile, and relocated it to the top folder alongside Makefile for easier access and build management • Added support for 32-bit Linux in the hab_log_parser tool • Removed OSX binaries; however, the build system and sources still support it • Added `BUILD.md` with instructions for building CST using Docker • Updated Code Signing Tool User Guide with instructions to use the PKCS#11 back end and notes about i.MX 9x devices
3.3.2	30 April 2023	• Added new back end, supporting HAB and AHAB signing using production keys stored on a hardware token • Removed the HSM back-end code, retaining only the PKCS#11 engine back end that is required for signing operations with the HSM • Updated srktool to support SHA-256 eFuse array
3.3.1	31 July 2020	• Built CST tool binaries using OpenSSL 1.1.1 • CST tool binaries support encrypted boot by default

All information provided in this document is subject to legal disclaimers.

© 2024 NXP B.V. All rights reserved.

Table 27. Software release history...*continued*

Release number	Release date	Change description
		• Added HABv4 log parser tool for Windows and macOS
3.3.0	31 December 2019	• Added support for macOS • Added AHAB signature block parser tool
3.2.0	30 April 2019	• Removed HAB3 support • Added encrypted boot support for AHAB • Added CST source code
3.1.0	31 August 2018	• Added OpenSSL 1.1.0 support • Added ECDSA support for HABv4 (only available in HAB 4.5.0) • Fixed bugs related to encrypted boot support • Added `convlb.exe` as a workaround for the line break limitation in Windows • Added HSM back end • Added HABv4 log parser tool • Added HABv4 CSF parser tool • Added HABv4 srktool script
3.0.1	11 May 2018	Fixed bugs related to Windows support
3.0.0	4 April 2018	Added support for AHAB
2.3.3	14 November 2017	• Added support for MS Windows • Removed support for the following commands: – Write Data – Clear Mask – Set Mask – Check Clear/Set – Set MID
2.3.2	15 March 2016	• Added support for manufacturing protection • Changed input from STDIN to a command-line argument • Made RNG unlock automatic only for CAAM
2.3.1	1 July 2015	Fixed a bug related to the 64-bit version of the SRK table
2.3	30 March 2015	Fixed bugs related to encrypted images
2.2	14 October 2014	• Added a note related to Linux RNG dependency • Added Section 7.1 having details related to replacing the CST back end • Corrected CA flag documentation
2.1	15 April 2013	Added support for HABv4 fast authentication
2.0	9 November 2012	Fixed bugs and made other updates
1.0	15 November 2011	Initial software release

All information provided in this document is subject to legal disclaimers.

© 2024 NXP B.V. All rights reserved.

12 Document revision history

Table 28 summarizes the revisions to this document.

Table 28. Document revision history

Document ID	Release date	Description
UG10106 v.4.0	5 December 2024	Aligned with Code Signing Tool release 4.0.0
UG10106 v.3.4.1	31 May 2024	Aligned with Code Signing Tool release 3.4.1
UG10106 v.3.4.0	31 December 2023	Aligned with Code Signing Tool release 3.4.0
UG10106 v.3.3.2	30 April 2023	Aligned with Code Signing Tool release 3.3.2
UG10106 v.3.3.1	31 July 2020	Aligned with Code Signing Tool release 3.3.1
UG10106 v.3.3.0	31 December 2019	Aligned with Code Signing Tool release 3.3.0
UG10106 v.3.2.0	30 April 2019	Aligned with Code Signing Tool release 3.2.0
UG10106 v.3.1.0	31 August 2018	Aligned with Code Signing Tool release 3.1.0
UG10106 v.3.0.1	11 May 2018	Aligned with Code Signing Tool release 3.0.1
UG10106 v.3.0.0	4 April 2018	Aligned with Code Signing Tool release 3.0.0
UG10106 v.2.3.3	14 November 2017	Aligned with Code Signing Tool release 2.3.3
UG10106 v.2.3.2	15 March 2016	Aligned with Code Signing Tool release 2.3.2
UG10106 v.2.3.1	1 July 2015	Aligned with Code Signing Tool release 2.3.1
UG10106 v.2.3	30 March 2015	Aligned with Code Signing Tool release 2.3
UG10106 v.2.2	14 October 2014	Aligned with Code Signing Tool release 2.2
UG10106 v.2.1	15 April 2013	Aligned with Code Signing Tool release 2.1
UG10106 v.2.0	9 November 2012	Aligned with Code Signing Tool release 2.0
UG10106 v.1.0	15 November 2011	Initial document release, aligned with Code Signing Tool release 1.0

All information provided in this document is subject to legal disclaimers.

© 2024 NXP B.V. All rights reserved.

Legal information

Definitions

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

Disclaimers

Limited warranty and liability — Information in this document is believed to be accurate and reliable. However, NXP Semiconductors does not give any representations or warranties, expressed or implied, as to the accuracy or completeness of such information and shall have no liability for the consequences of use of such information. NXP Semiconductors takes no responsibility for the content in this document if provided by an information source outside of NXP Semiconductors.

In no event shall NXP Semiconductors be liable for any indirect, incidental, punitive, special or consequential damages (including - without limitation - lost profits, lost savings, business interruption, costs related to the removal or replacement of any products or rework charges) whether or not such damages are based on tort (including negligence), warranty, breach of contract or any other legal theory.

Notwithstanding any damages that customer might incur for any reason whatsoever, NXP Semiconductors' aggregate and cumulative liability towards customer for the products described herein shall be limited in accordance with the Terms and conditions of commercial sale of NXP Semiconductors.

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

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

Applications — Applications that are described herein for any of these products are for illustrative purposes only. NXP Semiconductors makes no representation or warranty that such applications will be suitable for the specified use without further testing or modification.

Customers are responsible for the design and operation of their applications and products using NXP Semiconductors products, and NXP Semiconductors accepts no liability for any assistance with applications or customer product design. It is customer's sole responsibility to determine whether the NXP Semiconductors product is suitable and fit for the customer's applications and products planned, as well as for the planned application and use of customer's third party customer(s). Customers should provide appropriate design and operating safeguards to minimize the risks associated with their applications and products.

NXP Semiconductors does not accept any liability related to any default, damage, costs or problem which is based on any weakness or default in the customer's applications or products, or the application or use by customer's third party customer(s). Customer is responsible for doing all necessary testing for the customer's applications and products using NXP Semiconductors products in order to avoid a default of the applications and the products or of the application or use by customer's third party customer(s). NXP does not accept any liability in this respect.

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

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

Suitability for use in non-automotive qualified products — Unless this document expressly states that this specific NXP Semiconductors product is automotive qualified, the product is not suitable for automotive use. It is neither qualified nor tested in accordance with automotive testing or application requirements. NXP Semiconductors accepts no liability for inclusion and/or use of non-automotive qualified products in automotive equipment or applications.

In the event that customer uses the product for design-in and use in automotive applications to automotive specifications and standards, customer (a) shall use the product without NXP Semiconductors' warranty of the product for such automotive applications, use and specifications, and (b) whenever customer uses the product for automotive applications beyond NXP Semiconductors' specifications such use shall be solely at customer's own risk, and (c) customer fully indemnifies NXP Semiconductors for any liability, damages or failed product claims resulting from customer design and use of the product for automotive applications beyond NXP Semiconductors' standard warranty and NXP Semiconductors' product specifications.

HTML publications — An HTML version, if available, of this document is provided as a courtesy. Definitive information is contained in the applicable document in PDF format. If there is a discrepancy between the HTML document and the PDF document, the PDF document has priority.

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

Security — Customer understands that all NXP products may be subject to unidentified vulnerabilities or may support established security standards or specifications with known limitations. Customer is responsible for the design and operation of its applications and products throughout their lifecycles to reduce the effect of these vulnerabilities on customer's applications and products. Customer's responsibility also extends to other open and/or proprietary technologies supported by NXP products for use in customer's applications. NXP accepts no liability for any vulnerability. Customer should regularly check security updates from NXP and follow up appropriately.

Customer shall select products with security features that best meet rules, regulations, and standards of the intended application and make the ultimate design decisions regarding its products and is solely responsible for compliance with all legal, regulatory, and security related requirements concerning its products, regardless of any information or support that may be provided by NXP.

NXP has a Product Security Incident Response Team (PSIRT) (reachable at PSIRT@nxp.com) that manages the investigation, reporting, and solution release to security vulnerabilities of NXP products.

NXP B.V. — NXP B.V. is not an operating company and it does not distribute or sell products.

Trademarks

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

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

All information provided in this document is subject to legal disclaimers.

© 2024 NXP B.V. All rights reserved.

AMBA, Arm, Arm7, Arm7TDMI, Arm9, Arm11, Artisan, big.LITTLE, Cordio, CoreLink, CoreSight, Cortex, DesignStart, DynamIQ, Jazelle, Keil, Mali, Mbed, Mbed Enabled, NEON, POP, RealView, SecurCore, Socrates, Thumb, TrustZone, ULINK, ULINK2, ULINK-ME, ULINK-PLUS, ULINKpro, µVision, Versatile — are trademarks and/or registered trademarks of Arm Limited (or its subsidiaries or affiliates) in the US and/or elsewhere. The related technology may be protected by any or all of patents, copyrights, designs and trade secrets. All rights reserved.

EdgeLock — is a trademark of NXP B.V.

Microsoft, Azure, and ThreadX — are trademarks of the Microsoft group of companies.

Contents

Please be aware that important notices concerning this document and the product(s) described herein, have been included in section 'Legal information'.

© 2024 NXP B.V. All rights reserved.