

Encrypted Boot on HABv4 and CAAM Enabled Devices

1. Introduction

This application note describes the encrypted boot feature found in High-Assurance Boot (HAB) v4 supported devices, such as the i.MX 6 and i.MX 7 processors. Encrypted boot is a security mechanism that hides the sensitive information embedded in the bootloader data when it is in the persistent data storage.

Contents

1.	Introduction	1
1.1.	Purpose	2
1.2.	Intended audience and scope	2
1.3.	Definitions, Acronyms, and Abbreviations.....	2
2.	Overview	3
2.1.	HAB library.....	4
2.2.	Encrypted boot sequence	4
3.	Encrypted boot implementation.....	5
3.1.	Overview	5
3.2.	Requirements.....	6
3.3.	Protocol	6
3.4.	Data encryption key handling.....	7
3.5.	DEK blob.....	9
4.	Encrypted U-Boot example	10
4.1.	Assumptions	11
4.2.	Requirements.....	11
4.3.	Implementation.....	13
4.4.	Protecting the DEK blob after encrypted boot	19
5.	References	20
Appendix A.	Development scripts	20
A.1.	Building secure U-Boot.....	20
A.2.	Building signed U-Boot.....	21
A.3.	Building encrypted U-Boot	22
A.4.	Provisioning the PRIBLOB setting on the chip	25
Appendix B.	Troubleshooting	25
B.1.	Signed U-Boot CSF example	25
B.2.	Encrypted U-Boot CSF example	26
B.3.	Script to determine the DEK blob address	27



1.1. Purpose

The security suite in the i.MX 6 and i.MX 7 processors provides adequate features to establish a chain of trust for high-assurance computing. The security suite provides the capability to meet the trust computing requirements of embedded solutions, such as firmware data assurance.

Securing a hardware platform requires examination of its hardware components and verification of authenticity and integrity of the critical code that controls the platform. These security checks are executed after the reset by the boot code in the on-chip ROM. The ROM boot process (after verifying the authenticity of the bootloader script residing in the flash or external memory) follows the script to continue loading the operating system and data into the external memory. Because the bootloader script may reside in the external memory, it can be seen or used by unauthorized users.

The encrypted boot feature adds an extra security operation to the boot-loading sequence. It uses cryptographic techniques to obscure the bootloader data (it can be extended to the entire image), so it cannot be seen or used by unauthorized users. This mechanism protects and conceals the bootloader code residing in the flash (or external) memory. This application note describes the implementation of encrypted boot, which can be tailored to your requirements.

1.2. Intended audience and scope

This application note is intended for technically qualified users who are familiar with *Secure Boot on i.MX 50, i.MX 53, i.MX 6 and i.MX 7 Series using HABv4* (document [AN4581](#)) and can successfully deploy secure boot on HABv4-enabled i.MX processors.

This application note describes the encrypted boot feature added to the HAB and the Code Signing Tool (CST). Encrypted boot is only featured on these applications processors from the i.MX family:

- i.MX 6—6Dual, 6Quad, 6DualPlus, 6Quad Plus, 6DualLite, 6Solo, 6SoloX, 6UltraLite.
- i.MX 7—7Solo, 7Dual, 7ULP (this device is in the pre-production phase when this document is published).

This document only demonstrates the encrypted boot solution on the i.MX 6 and i.MX 7 processors. The internals of the encrypted boot require an extensive knowledge of cryptography and security trust models, which is out of the scope of this document. For more details about the internals, see the i.MX 6 (or i.MX 7) security reference manual.

This document focuses on:

- Encrypted boot requirements.
- Encrypted boot components.
- Encrypted boot image generation.

1.3. Definitions, Acronyms, and Abbreviations

The terms and acronyms used in this document are:

- AES—Advanced Encryption Standard.
- CA—Certificate Authority; the holder of a private key used to certify public keys.
- CAAM—Cryptographic Acceleration and Assurance Module; an accelerator for encryption,

stream cipher, and hashing algorithms, with a random number generator and run-time integrity checker.

- CMS—Cryptographic Message Syntax; a general format for data that may have cryptography applied to it, such as digital signatures and digital envelopes. HAB uses the CMS as a container to hold the PKCS#1 signatures.
- CSF—Command Sequence File; a binary data structure interpreted by the HAB to guide authentication operations.
- CST—Code Signing Tool; an application running on the build host to generate the CSF and associated digital signatures.
- DCD—Device Configuration Data; a binary table used by the ROM code to configure the device at early boot stages.
- DEK—Data Encryption Key; the AES key used to encrypt/decrypt the boot image.
- HAB—High-Assurance Boot; a software library executed in the internal ROM of NXP processors at boot time which, among other things, authenticates the software in the external memory by verifying digital signatures according to the CSF.
- IVT—Image Vector Table.
- OS—Operating System.
- OTP—One-Time Programmable. The OTP hardware includes the masked ROM and electrically programmable fuses (eFuses).
- PKCS#1—A standard that specifies the use of the RSA algorithm.
- PKI—Public Key Infrastructure; a hierarchy of public key certificates in which each certificate (except for the root certificate) can be verified using the public key above it.
- RSA—A public key cryptography algorithm developed by Rivest, Shamir, and Adleman.
- SA—Signature Authority; the holder of the private key used to sign software components.
- SDP—Serial Download Protocol; also called UART/USB serial download mode. It allows code provisioning through UART or USB during the production and development phases.
- SRK—Super Root Key; an RSA key pair which forms the start of the boot-time authentication chain. The hash of the SRK public key is embedded in the processor using OTP hardware. The SRK private key is held by the CA. Unless explicitly noted, the SRK acronym (in this document) refers to the public key only.

2. Overview

It is recommended to read *Secure Boot on i.MX 50, i.MX 53, i.MX 6 and i.MX 7 Series using HABv4* (document [AN4581](#)) before reading this document. Document [AN4581](#) provides technical overview of the secure boot feature and the signing procedure of image binaries required to understand the use cases and processes described later on in this document.

2.1. HAB library

The HAB library is a ROM component and contains security mechanisms, such as the authentication, encryption, and decryption operations. The secure boot sequence allows the ROM code to use the HAB library to enforce cryptographic checks at each booting stage. This process provides the foundations for a secure environment by asserting the integrity of the software images that are about to execute. These cryptographic checks prevent any unauthorized software from running on the target. The same library calls can be accessed at later boot stages to extend the trust chain past the ROM-booting stage.

At boot time, the Command Sequence File (CSF) issues a series of HAB commands to complete the secure boot sequence. Therefore, the security checks at this stage are specific to the provided CSF. The CSF starts as a human-readable text file which is processed by the Code Signing Tool (CST) and compiled into a ROM-readable binary file. Generally, these commands include (but are not limited to) the authentication and decryption of memory locations. The CSF processing executed by the HAB library includes PKI operations, cryptographic hashing, and digital signature verification. The HAB library may use the on-board hardware accelerators to improve the boot performance and access the OTP master keys.

NOTE

Encrypted boot is only supported on certain i.MX processors with HAB version 4.1 (or higher) that use the CAAM cryptographic accelerator engine.

NOTE

For devices with HAB 4.4.0 (or lower), the HAB code locks the job ring and DECO master ID registers in a closed configuration. If the user-specific application requires any changes in the CAAM MID registers, it is necessary to add the “Unlock CAAM MID” command into the CSF file. The current NXP BSP implementation expects the CAAM registers to be unlocked when configuring the CAAM to operate in the non-secure TrustZone world. This applies when OP-TEE is enabled on the i.MX 6, i.MX 7, and i.MX 7ULP processors.

2.2. Encrypted boot sequence

The security solution for an embedded system starts as early as the boot ROM execution. Secure boot extends the ROM execution flow for HABv4-enabled processors; encrypted boot extends it even further. The extended boot flow is shown in [Figure 1](#). Note that the “Process CSF” is extended beyond the digital signature verification. The encrypted boot solutions include the addition of a cryptographic operation required to decrypt the encrypted image data. When configured for the encrypted boot operation, the boot ROM on these devices does not allow unauthenticated (nor modified) code to execute. Any failure or security violation generated during the CSF processing halts the boot process when the CSF processing ends.

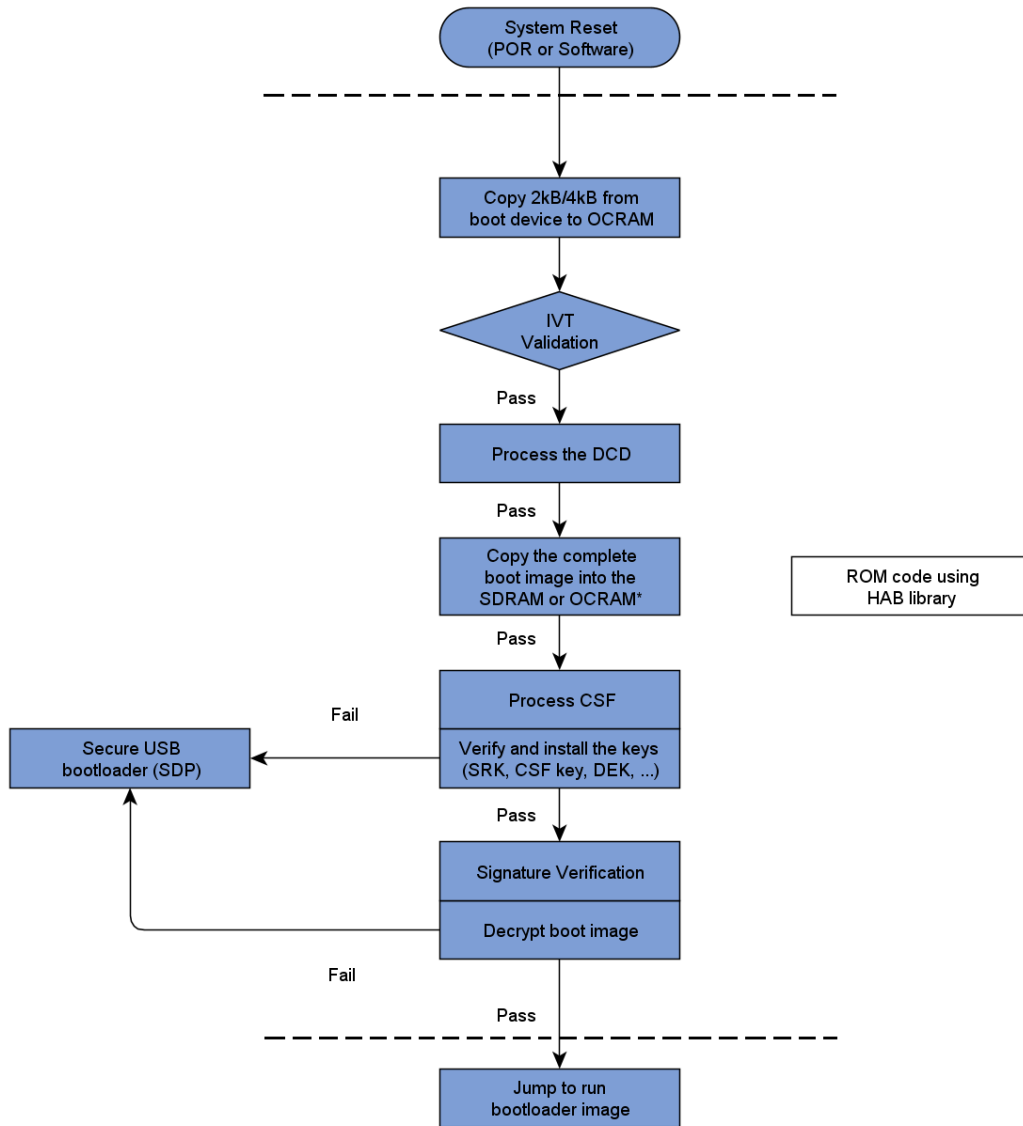


Figure 1. Encrypted boot flow from the boot device

3. Encrypted boot implementation

3.1. Overview

Simply put, encrypted boot is a secure-boot version of an encrypted bootloader. Therefore, this protocol can be divided into two protection mechanisms. The first mechanism is the digital signature, which authenticates the source of the binary image. The second mechanism is the bootloader code encryption, which bestows confidentiality to the bootloader data. Keep in mind that encrypted boot uses both mechanisms in any order (sign and then encrypt or encrypt and then sign) and both operations are applied on the same region (except the header in plaintext, which can only be signed but not encrypted).

3.2. Requirements

As described in the respective i.MX applications processor reference manual, the ROM requires a preset number of data structures from the program image. In addition to these structures, the encrypted boot image has two more data structures. These are the CSF and the DEK blob. The encrypted boot image consists of:

- Image Vector Table (IVT)—a table of address pointers used by the boot ROM to locate other required data components.
- Boot data structure—a simple structure indicating where to load the boot image and specifying the size of the image.
- Device Configuration Data (DCD)—a list of registers that the ROM programs with the provided data to perform an early initialization of the system. The DCD is typically used to initialize the SDRAM.
- Command Sequence File (CSF) and the associated data—a block of data that contains the commands that the HAB executes during the boot, as well as the associated certificates and signatures that the HAB uses to verify images.
- Data Encryption Key blob (DEK blob)—the DEK is used to decrypt the encrypted boot image. The DEK blob is used as a security layer to wrap and store the DEK off-chip, which is unique to the chip that generated the blob.

Only the DEK blob generation and encryption process is described in this application note. For more information about other components and the digital signature process, see *Secure Boot on i.MX 50, i.MX 53, i.MX 6 and i.MX 7 Series using HABv4* (document [AN4581](#)).

3.3. Protocol

The encrypted boot protocol is an advanced use of the secure boot protocol. Encrypted boot is a combination of digital signature verification and decryption of the bootloader image code. These steps are a simplification of the encrypted boot protocol:

1. Signing a boot image.
2. Encrypting and signing of a boot image, or signing and encrypting of a boot image.
3. DEK blob generation and encrypted boot image assembling.

The first and second steps involve signing a binary image; this procedure follows the instructions from *Secure Boot on i.MX 50, i.MX 53, i.MX 6 and i.MX 7 Series using HABv4* (document [AN4581](#)). Keep in mind that the authenticated boot image in step 1 is independent from the encrypted boot image in step 2. For both steps, the digital signature authentication follows the same procedure as the secure boot. As specified in *Secure Boot on i.MX 50, i.MX 53, i.MX 6 and i.MX 7 Series using HABv4* (document [AN4581](#)), a unique set of keys is generated by the CST, and these keys are used to create the digital signature. During the process, a Super Root Key (SRK) table and certificates are also generated. Because the keys and certificates are unique to the target processor, the same set is used for the encrypted boot.

In step 2, the boot data is encrypted and a digital signature of the whole boot data image including the boot data structure (or vice versa) is generated. Note that the boot data structures do not contain any confidential data and are used by the ROM before the CSF is processed. Therefore, these data structures

remain in plaintext and are included in the digital signature. This allows the boot ROM to access the necessary pointers to initialize the data structures and modules required by HAB.

The encryption part of step 2 is straightforward. By adding two new commands to the CSF, the CST continues to generate the digital signature and produces a DEK to encrypt the boot image. [Figure 2](#) shows the process of the boot image encryption.

The third step generates a secure blob using on-chip private keys. This security measure ensures that this specific chip is the only chip that can encrypt and decrypt the blob. However, CAAM is the only component with access to the OTP master keys. NXP provides specialized tools for DEK blob generation through the NXP U-Boot port. Finally, the CSF and the DEK blob are appended to the encrypted image.

The following sections describe steps 2 and 3 in more detail. For a step-by-step guide of step 1, see *Secure Boot on i.MX 50, i.MX 53, i.MX 6 and i.MX 7 Series using HABv4* (document [AN4581](#)).

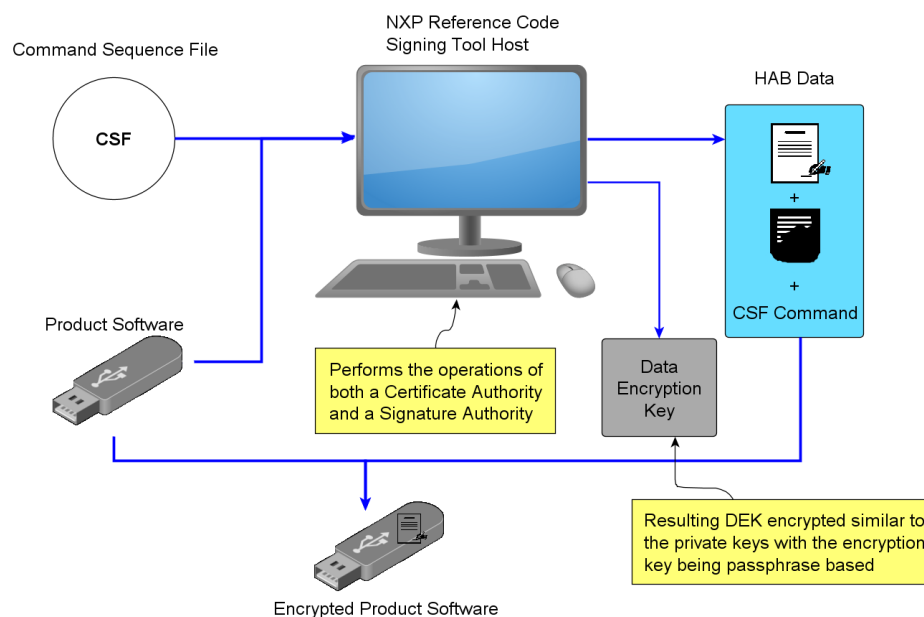


Figure 2. Encrypting image using CST

3.4. Data encryption key handling

The CST automatically generates a random DEK when the “Install Secret Key” command is present in the CSF and the CST is in the encryption mode. After the digital signature is authenticated, the ROM proceeds to decrypt the encrypted bootloader using the DEK. Unlike the RSA public-key system used for the authentication, the encrypted boot uses symmetric-key algorithms, specifically multiple key-length variants of the Advanced Encryption Standard (AES) algorithm. Therefore, the DEK sensitivity requires for it to be protected at all times (including manufacturing) and especially in the end product. The following sections address both cases.

3.4.1. Protection for the end product

A unique OTP Master Key (OTPMK) is used to encrypt and wrap the DEK in a blob. The OTPMK is protected by the hardware and can be accessed only by CAAM. Consequently, this step must be executed on a target processor with a software capable of using CAAM. NXP provides a freely available U-Boot port with such capability.

The fact that the OTPMK can only be accessed by CAAM means that the blob can only be decrypted by the same processor that encrypted it. To further add to the security of the DEK, the blob is decapsulated and decrypted inside a secure memory partition that can only be accessed by CAAM. It is also highly recommended to increment the PRIBLOB setting in CAAM to make this blob undecipherable by any software that runs after the encrypted boot. This way the DEK blob remains private to the instance of the encrypted boot and secure from any attacks to extract the DEK. For more information about the PRIBLOB, see the Security Reference Manual (SRM) of the respective NXP chipset.

3.4.2. Protection for manufacturing

The CST asks for a public key to encrypt the resulting DEK. This provides a protection layer for the DEK in the host machine. This layer of security prevents any mishandling of the DEK off-chip; for example, when transferring it from one provider to another. The following steps show a way to generate the key pair using the OpenSSL command-line utility. The generated self-signed certificate is then used by the CST to encrypt the DEK, which then can be safely transferred through a non-secure channel of communication.

- Generate a 2048-bit RSA key pair:
`openssl genrsa -out ./dek_rsa_key.pem 2048`
- Generate a certificate-signing request with the key pair:
`openssl req -new -key ./dek_rsa_key.pem -out ./dek_rsa_key.csr`
- Generate a self-signed certificate:
`openssl x509 -req -days 365 -in ./dek_rsa_key.csr -signkey ./dek_rsa_key.pem -out ./dek_rsa_key.crt.pem`

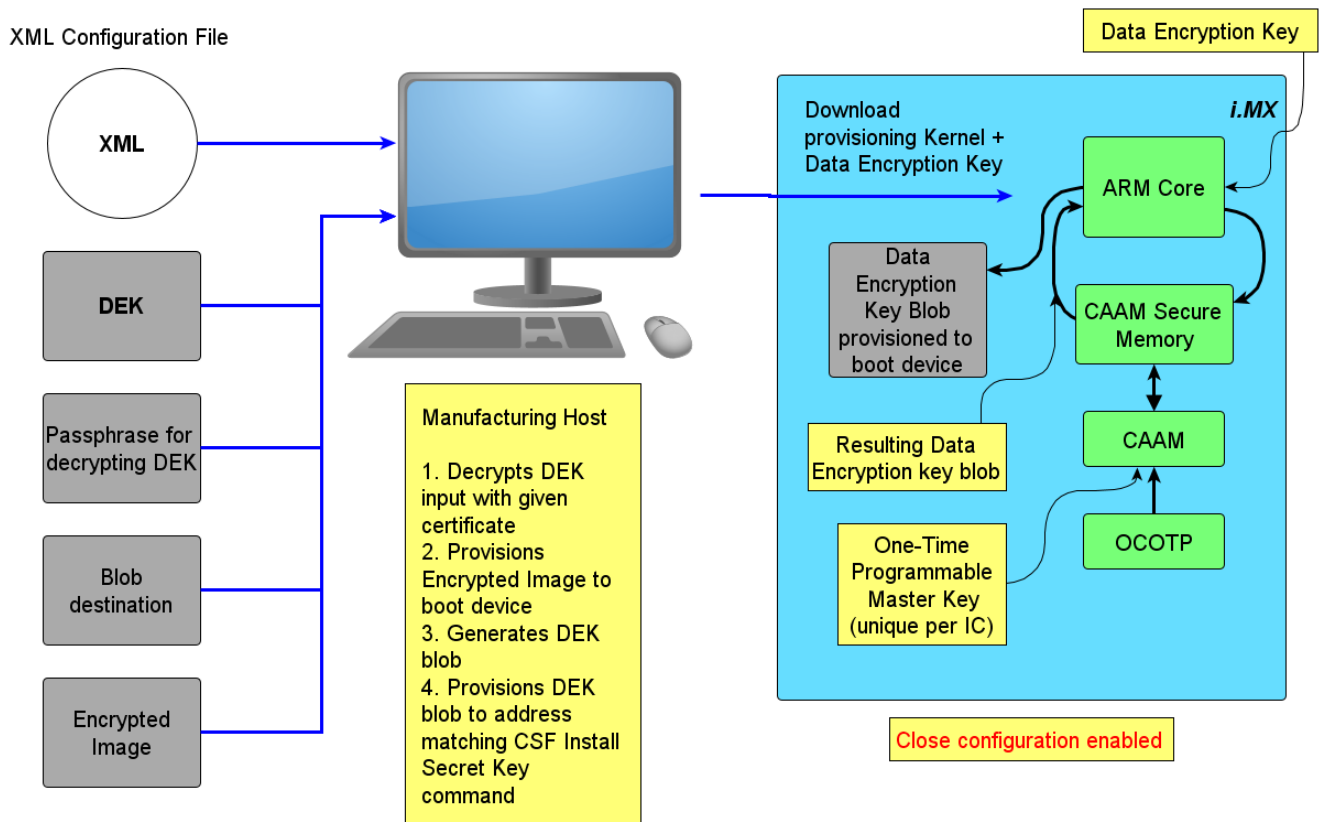


Figure 3. DEK provisioning for manufacturing

The previous security layer anticipates the use of a contract manufacturer by the OEM. A good security practice is to never share passwords. Therefore, the intent for the contract manufacturer is to provide the OEM with a public key. The OEM encrypts the DEK and sends it to the manufacturer, which can decrypt the DEK with the corresponding private key. If a contract manufacturer is not used, then the CA key (generated for the authenticated boot by the CST) can be used for this purpose.

Because the DEK generated by the CST is in plaintext, the OEM can operate on it. If the DEK is encrypted by the public key generated above, the DEK can be obtained using this OpenSSL command:

```
openssl cms -decrypt -in dek_in.bin -inform DER -out dek_out.dec -binary -inkey private_key.pem
```

Where:

- `dek_in.bin` is the DEK protected with the public key `dek_rsa_key.pem`.
- `dek_out.bin` is the plaintext DEK.
- `private_key` is the private key corresponding to the public key given to the CST.

3.5. DEK blob

The DEK blobs are stored by the HAB in a secret key blob data structure, which is shown in [Figure 4](#). For more details, see the *High-Assurance Boot Version 4 Application Programming Interface Reference Manual*.

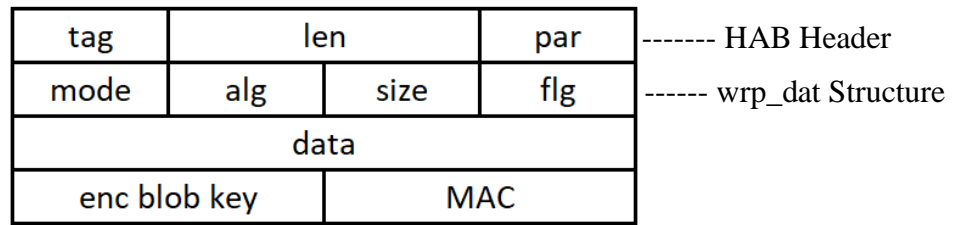


Figure 4. Key blob data structure

Parameters:

- tag: constant-identifying HAB data structure HAB_TAG_WRP = 0x81
- len: structure length in 8-bit
- par: constant value HAB Version = 0x41
- mode: constant value HAB_MODE_CCM = 0x66
- alg: constant value HAB_ALG_AES = 0x55
- size: unwrapped key value size in bytes
- flg: key flags
- data: encrypted key value
- enc blob key: encrypted blob key
- MAC: Message Authentication Code

The DEK blob utility builds this structure from a given DEK. Notice that the HAB supports a set of encryption algorithms, but the encrypted boot protocol expects AES. The key length is a variable; it can be 128-bit, 192-bit, or 256-bit.

4. Encrypted U-Boot example

This example uses the U-Boot image as the bootloader. U-Boot is commonly used as a bootloader for Linux® OS devices and provided by the NXP Linux BSP.

The default memory layout of the NXP U-Boot port can be modified to meet the encrypted boot requirements. This is shown in [Figure 5](#). This layout is similar to any other U-Boot port, with the addition of the security-related data appended at the end of the image.

Memory Layout for secure U-Boot image
CONFIG_SECURE_BOOT

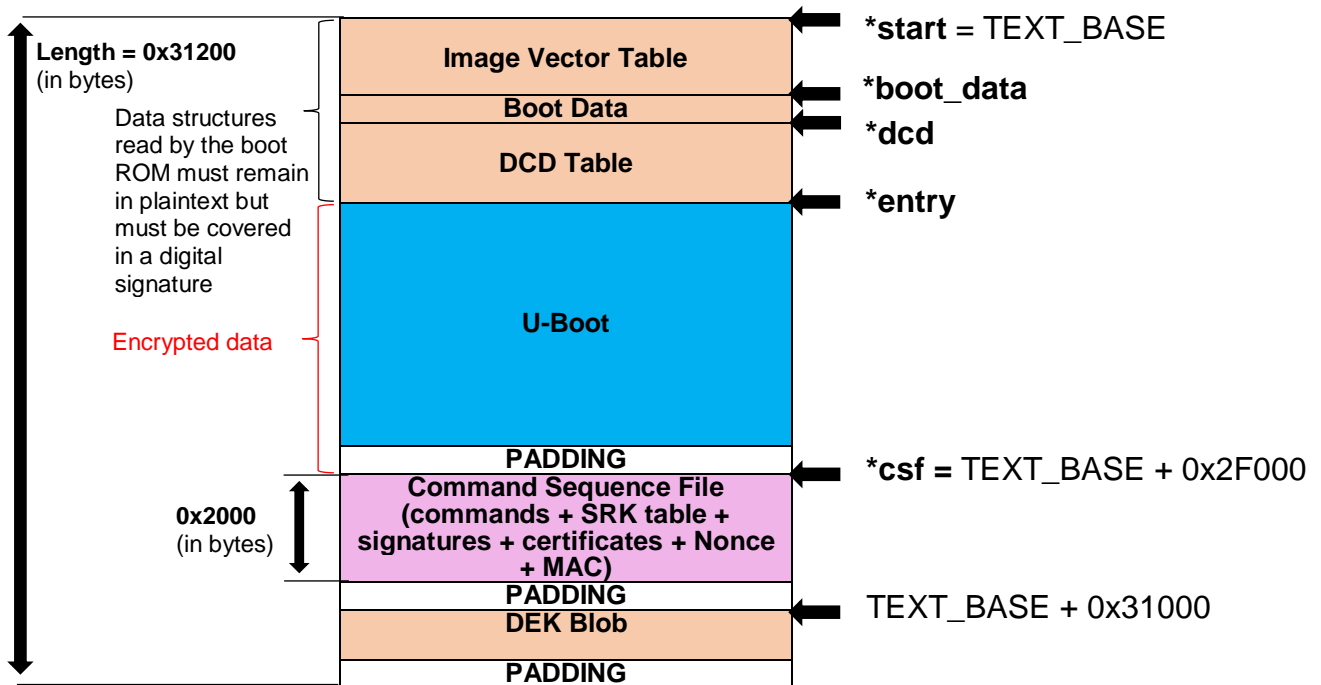


Figure 5. Memory layout of the encrypted U-Boot

4.1. Assumptions

When designing the U-Boot image as an encrypted boot solution, there are three assumptions which accelerate and simplify the construction process.

- The U-Boot image can be built for multiple board configurations. For demonstration purposes, this example uses the i.MX 6Quad processor. For other boards, use the associated configuration file to build the U-Boot and see the reference manual of the particular chip to write the fuses.
- The user is familiar with the secure configuration for the U-Boot and can properly sign and boot a U-Boot image.
- The encrypted image is constructed by an individual party, and there is no need to worry about provisioning the DEK.

4.2. Requirements

The components required to build an encrypted image are shown below. Note that the majority of these components are the product of this U-Boot image signing procedure:

- The OpenSSL library is installed:
 - `sudo apt-get install libssl-dev`
- The CST is in the encryption mode:
 - To build the CST in the encryption mode, run these commands:

–To relink it on 32-bit machines:

```
cd <CST install directory>/code/back_end/src
gcc -o cst -I ../hdr -L ../../../../linux32/lib *.c -lfrontend -lcrypto <-m32 for 64-bit machines>
cp cst ../../../../linux32/bin
```

–To relink it on 64-bit machines:

```
cd <CST install directory>/code/back_end/src
gcc -o cst -I ../hdr -L ../../../../linux64/lib *.c -lfrontend -lcrypto
cp cst ../../../../linux64/bin
```

NOTE

The CST is not in the encryption mode by default. This feature must be enabled before encrypting the bootloader image. The performance of the CST may be affected due to its dependency on the host entropy. See the *CST User Guide* for more details.

- The i.MX 6Quad SabreSD device is in a secure mode.
- The U-Boot image with the secure boot and encrypted boot support (with the blob generation tool) is enabled:
 - To configure the U-Boot to be built with secure boot support, CONFIG_SECURE_BOOT must be defined in the board header file:
 - For older U-Boot versions (v2016.09 and lower): Uncomment or add CONFIG_SECURE_BOOT to the board configuration header (in *include/configs/mx6sabre_common.h*).
 - Newer U-Boot versions (since v2016.11): Add CONFIG_SECURE_BOOT=y into the board configuration file (in *configs/mx6qsabresd_defconfig*) or select it in the U-Boot configuration menu:
 - ➔ Arm architecture ➔ Check “Support i.MX HAB features”.
 - To configure the U-Boot to be built with encrypted boot support, follow these steps (depending on the BSP release):
 - imx_v2016.03_04.1.15_2.0.0_ga and older:** define these configurations in the board header file (in *include/configs/mx6sabre_common.h*):
 - CONFIG_SYS_FSL_SEC_COMPAT 4 /* HAB version */.
 - CONFIG_FSL_CAAM.
 - CONFIG_CMD_DEKBLOB.
 - CONFIG_SYS_FSL_LE.
 - imx_v2017.03_4.9.11_1.0.0_ga:** Select CONFIG_SECURE_BOOT in *Kconfig/defconfig*.
 - imx_v2017.03_4.9.88_2.0.0_ga and newer:** Select CONFIG_CMD_DEKBLOB in *Kconfig/defconfig*.
- Signed U-Boot image:
 - U-Boot image with the CSF and digital signature attached.

4.3. Implementation

Many different implementations to construct an encrypted U-boot image are possible. The right implementation depends on the solution's requirements. The presented implementation provides the foundation principles; it can be modified to meet different needs. [Appendix A, "Development scripts"](#) provides a list of scripts to automate this implementation.

4.3.1. CSF for encryption

The CSF contains several commands, which are described in detail in the *High-Assurance Boot CST User Guide*. The CSF for encrypted boot is similar to the secure-boot CSF, with the addition of two commands: *[Install Secret Key]* and *[Decrypt Data]*.

4.3.1.1. [Install Secret Key]

The DEK must be installed in the key storage for the CAAM to use it. The *[Install Secret Key]* command specifies the DEK properties and where to install it in the key storage.

NOTE

The key storage used for the DEK is independent from the SRK storage. Therefore, there is no conflict with duplicated key indices. Keep track of the indices for the secret key storage after they are overwritten.

CAAM uses the AES-CCM algorithm for the encrypted boot. Therefore, the key length specified in this section defines which variant to use. This command is an example of using AES-192:

```
[Install Secret Key]
    Verification index = 0
    Target Index = 0
    Key = "dek.bin"
    Key Length = 192
    Blob Address = 0x17857000
```

- Verification index—the index of the master key used in the encryption algorithm.
- Target index—the index of the key storage that holds the new key.
- Key—path to the DEK file generated by the CST.
- Key length—the length of the key for the AES algorithm. Possible lengths are 128-bit, 192-bit, and 256-bit.
- Blob address—the address in RAM where the DEK blob is loaded.

It is recommended to append the blob to the U-Boot image in the same way as the CSF was appended. The reason for padding the CSF is that the blob can be stored at a known location, which is used to compute the blob address in the RAM.

NOTE

This implementation forces the total size of the encrypted image (boot data structures, U-Boot code, CSF, and DEK blob) to be lesser than (or equal to) the image size specified in the boot data.

The DEK blob is placed at the end of the padded CSF in the final encrypted boot image. Thus, the blob address can be calculated as follows:

```
Blob Address = IVT start address + u-boot size + Padded CSF size (0x2000)
IVT start address = 0x177ff400
u-boot size = 0x55c00
```

4.3.1.2. [Decrypt Data]

This command specifies the data range being decrypted by CAAM. The command defines the index of the key, which is specified in the *[Install Secret Key]* command. In addition, CAAM computes the Message Authentication Code (MAC) of the encrypted DEK, which is different from the MAC generated after the boot image is encrypted (they are called BLOB_MAC and IMAGE_MAC in the following sections). The BLOB_MAC is appended in the blob and intended to check the integrity of the encrypted DEK. This is just one more layer of security provided by CAAM. Lastly, CAAM decrypts the data in place, meaning that the data is decrypted into the same place in RAM where it was found.

```
[Decrypt Data]
Verification index = 0
Mac Bytes = 16
Blocks = <address> <offset> <size> <file>,
        0x17800000 0xc00 0x55000 "u-boot-dtb.imx"
```

- Verification index—the index of the master key used in the encryption algorithm.
- Mac bytes—the length of BLOB_MAC; supported lengths are 4, 8, and 16 bytes.
- <address>—the location in the device's RAM where the data are loaded.
- <offset>—the offset into the image file to start the decryption.
- <size>—the size of the decrypted data.
- <file>—the path to the image file.

With the current implementation of the CST, it is not possible to construct a CSF that can sign the encrypted boot image which then can be authenticated by the HAB before decryption, because the HAB interprets the CSF commands serially (and the other way round—encrypt a signed boot image). Due to this constraint, the CSF is split into two:

- Case 1, *encrypt signed U-Boot*—the first one is used to sign the boot image and the second one is used to encrypt the signed image (except for the header) and create the DEK.
- Case 2, *sign encrypted U-Boot*—the first one is used to encrypt the boot image and create the DEK and the second one is used to sign the encrypted image including the boot data structures.

4.3.1.3. Signing and encrypting the signed image

CSF to sign the boot image (CSF-Sign):

[Authenticate Data]

```
Verification index = 2
Blocks = 0x177ff400 0x000 0x55c00 "u-boot-dtb.imx"
        <address> <offset> <size> <file>
```

- <address>—the location of the image loaded into the device's RAM.
- <offset>—the offset into the image file to start signing.
- <size>—the size to sign in the image file.
- <file>—the path to the image file.

CSF to encrypt the signed image (CSF-Enc-Sign):

The CSF for encrypted boot is similar to the secure-boot CSF, with the addition of two commands and a slight modification of the [Authenticate Data] command.

[Authenticate Data]

```
Verification index = 2
Blocks = 0x177ff400 0x000 0xc00 "u-boot-dtb.imx"
        <address> <offset> <size> <file>
```

- <address>—the location of the image loaded into the device's RAM.
- <offset>—the offset into the image file to start signing.
- <size>—the size to sign in the image file, size = [boot image size] – [u-boot padded code size].
- <file>—the path to the image file.

Two new commands are added to encrypt the boot image and create a DEK and to provide the DEK blob location to decrypt.

[Install Secret Key]

```
Verification index = 0
Target Index = 0
Key = "dek.bin"
Key Length = 192
Blob Address = 0x17857000
```

- Verification index—the index of the master key used in the encryption algorithm.
- Target index—the index of the key storage to hold the new key.
- Key—the path to the DEK file generated by the CST.
- Key length—the length of the key for the AES algorithm. The possible lengths are 128-bit, 192-bit, and 256-bit.

- Blob address—the address in RAM where the DEK blob is loaded.

[Decrypt Data]

```
Verification index = 0

Mac Bytes = 16

Blocks = <address> <offset> <size> <file>,
        0x17800000 0xc00    0x55000 "u-boot-dtb.imx"
```

- Verification index—the index of the master key used in the encryption algorithm.
- Mac bytes—the length of the MAC. The supported lengths are 4, 8, and 16 bytes.
- <address>—the location where the data is loaded into the device’s RAM.
- <offset>—the offset into the image file to start decrypting.
- <size>—the size to decrypt the data.
- <file>—the path to the image file.

4.3.1.4. Encrypting and signing the encrypted image

CSF to encrypt boot image and create DEK (CSF-Enc):

Because the boot data structures do not contain any secret data and are used by the ROM before the CSF is processed, these data structures remain in plaintext and are included in the digital signature using the following modified [Authenticate Data] command.

[Authenticate Data]

```
Verification index = 2

Blocks = 0x177ff400 0x000    0xc00 "u-boot-dtb.imx"
        <address> <offset> <size> <file>
```

- <address>—the location of the image loaded into the device’s RAM.
- <offset>—the offset into the image file to start signing.
- <size>—the size to sign in the image file, size = [boot image size] – [u-boot padded code size].
- <file>—the path to the image file.

The rest of the boot image is encrypted using the [Decrypt Data] command mentioned above.

CSF to sign the encrypted image (CSF-Sign-Enc):

A digital signature must contain the whole boot data image, which consists of the boot data structures and the encrypted boot image. This ensures that the encrypted image cannot be swapped with a malicious image even if the DEK is compromised, because the attacker would need to first match the digital signature generated from the signed encrypted image. In this step, the [Authenticate Data] command is the same as the command for the secure boot process.

[Authenticate Data]

```
Verification index = 2

Blocks = 0x177ff400 0x000    0x55c00 "u-boot-dtb.imx"
        <address> <offset> <size> <file>
```

- <address>—the location of the image loaded into the device’s RAM.

- <offset>—the offset into the image file to start signing.
- <size>—the size to sign in the image file.
- <file>—the path to the image file.

Because this CSF is used by the HAB to authenticate and decrypt the encrypted boot image, it must contain the dummy [Install Secret Key] and [Decrypt Data] commands. The DEK and the encrypted boot image created by this CSF are discarded (in bold below).

[Install Secret Key]

```
Verification index = 0
Target Index = 0
Key = "dek.bin.dummy"
Key Length = 192
Blob Address = 0x17857000
```

- Verification index—the index of the master key used in the encryption algorithm.
- Target index—the index of the key storage to hold the new key.
- Key—the path to the DEK file generated by the CST.
- Key length—the length of the key for the AES algorithm. The possible lengths are 128-bit, 192-bit, and 256-bit.
- Blob address—the address in RAM where the DEK blob is loaded.

[Decrypt Data]

```
Verification index = 0
Mac Bytes = 16
Blocks = <address> <offset> <size> <file>,
        0x17800000 0xc00 0x55000 "u-boot-dtb.imx.dummy"
```

- Verification index—the index of the master key used in the encryption algorithm.
- Mac bytes—the length of the MAC; the supported lengths are 4, 8, and 16 bytes.
- <address>—the location where the data is loaded into the device's RAM.
- <offset>—the offset into the image file to start decrypting.
- <size>—the size to decrypt the data.
- <file>—the path to the image file.

When the [Decrypt Data] command is compiled by the CST, it encrypts the data blocks in place of the given files using DEK and generates the IMAGE_MAC data which is appended to the CSF. The CST also generates the nonce used for the encryption. Because only the CSF binary generated by this CSF is used by the HAB, the Nonce/IMAGE_MAC data must be swapped with the one from the first CSF because the encrypted image and the DEK from the first CSF are used. This involves another step in the encrypted boot process before the encrypted boot image is assembled for booting onto the chip.

Both CSFs for this example are shown in [Appendix B.2, “Encrypted U-Boot CSF example”](#).

4.3.2. U-Boot encrypted memory layout

Perform these steps to generate the layout defined above for the encrypted and signed U-Boot example:

1. Generate the CSF for the boot image encryption, similar to that in [Appendix B.2, “Encrypted U-Boot CSF example”](#). The address and offset depend on the generated keys and the size of *u-boot-dtb.imx*.
2. (Optional) Generate a RSA key pair to protect the DEK off-chip with the OpenSSL commands shown in [Section 3.4.2, “Protection for manufacturing”](#).

This example uses a copy of a key pair from the HABv4 PKI tree.

```
cp crts/IMG1_1_sha256_2048_65537_v3_usr.crt.pem dek_rsa_key.crt.pem
```

3. Encrypt and sign the U-Boot image by providing the CSF to the CST.

If an encrypted DEK is needed:

```
./cst -o csf_enc.bin -c ./dek_rsa_key.crt.pem -I u-boot_enc.csf
```

If a plaintext DEK is needed:

```
./cst--o csf_enc.bin--i u-boot_enc.csf
```

NOTE

At this point, the *u-boot-dtb.imx* file is almost entirely replaced by its encrypted version and the *dek.bin* file is created (plaintext/encrypted) in the given path.

4. Generate the CSF to sign the encrypted boot image, similar to that in [Appendix B.2, “Encrypted U-Boot CSF example”](#). The address and offset depend on the generated keys and the size of the *u-boot-dtb.imx* file.
5. Sign the encrypted U-Boot including the boot data structures from Step 3 and encrypt a dummy U-Boot by providing the CSF to the CST.

```
./cst --o csf_sign_enc.bin --i u-boot_sign_enc.csf
```

6. Swap the Nonce/IMAGE_MAC from *csf_enc.bin* to *csf_sign_enc.bin*.

— Calculate the Nonce/IMAGE_MAC size based on the MAC bytes’ value in the CSF.

Mac Bytes = 16, thus,

Nonce/MAC size = 12 (Nonce size) + 16 (MAC bytes) + 8 (CSF header for Nonce/MAC) = 36

— Calculate the Nonce/IMAGE_MAC offset in the CSF.

MAC offset = *csf_enc.bin* size - Nonce/MAC size = 3972 - 36 = 3936

— Extract Nonce/IMAGE_MAC from *csf_enc.bin*.

```
dd if=csf_enc.bin of=noncemac.bin bs=1 skip=3936 count=36
```

— Replace IMAGE_MAC from *csf_enc.bin* to *csf_sign_enc.bin*.

```
dd if=noncemac.bin of=csf_sign_enc.bin bs=1 seek=3936 count=36
```

7. Pad the CSF to 0x2000. The reason is that it provides a known address for the blob to be stored to. Check that the known address is the same as that specified in the CSF.

```
objcopy -I binary -O binary --pad-to 0x2000 --gap-fill=0xff csf_sign_enc.bin csf_sign_enc_padded.bin
```

8. Assemble *u-boot_encrypted_no_dek.bin*, which is the append of the U-Boot image and the obtained CSF:

```
cat u-boot-dtb.imx csf_sign_enc_padded.bin > u-boot_encrypted_no_dek.bin
```

9. (Optional) If *dek.bin* is obfuscated for manufacturing purposes, the DEK must be decrypted. The U-Boot port provides the tools to generate a DEK blob using a DEK in plaintext. To use the

U-Boot utility, the DEK must be decrypted by this command:

```
openssl cms -decrypt -in dek.bin -inform DER -out dek_paintext.bin -binary -inkey
private_key.pem
```

10. Boot the device using the signed and unencrypted U-Boot. There are different ways to load the DEK into the board, such as TFTP or loading to the SD card. When the DEK is loaded to the RAM, run this U-Boot command:

```
dek_blob <DEK address> <Blob Address> <key length>
```

The output looks like this:

```
=> dek_blob 0x17870000 0x17871000 192
Generating DEK blob using key at
0x17870000
28 EB 58 03 ED 70 00 23 A9 58 AA 8F E8 10 52 27 4B 3F E7 0E C3 4F CF D5

DEK Blob is available at 0x17871000
810050416655180072B5556654639EDAA98B68FE931376301F207140110C7EC5BD898
62DF05218BECA18FEE72281667D235765B19E1FCB7DDD7FA2D9E1E145CA4D1EFCCDA3
929B6BE7C40240757FB7AD
```

This command takes three parameters as an input. The first parameter is the location of the DEK. The second parameter is the location to store the DEK blob to. The third parameter is the length of the key for the AES algorithm.

The command stores the blob to the indicated address and also prints it to the console.

11. Copy or transfer the DEK from the device to the host where the encrypted image resides. This can be done using TFTP (or HEX) editors, such as Hexedit. In this case, the file is named *dek_blob.bin*.
12. Pad the *u-boot_encrypted_no_dek.bin* file to the size of the unsigned U-Boot + 0x2000. The reason is that it ensures that the DEK blob is at the same address as that in the CSF.

```
objcopy -I binary -O binary --pad-to 0x57c00 --gap-fill=0x00 u-
boot_encrypted_no_dek.bin u-boot_encrypted_no_dek_padded.bin
```

13. Append the DEK blob to the encrypted U-Boot image:

```
cat u-boot_encrypted_no_dek_padded.bin dek_blob.bin > u-boot_encrypted.bin
```

14. Boot the device using *u-boot_encrypted.bin*.

4.4. Protecting the DEK blob after encrypted boot

DEK blobs are an integral part of the encrypted boot process used to derive the DEK needed to decrypt and boot the image. When designing the encrypted boot using the DEK blob, it is necessary to inhibit any modification or replacement of the DEK blob with a counterfeit one to prevent execution of malicious code. The PRIBLOB setting in the CAAM allows the secure boot software to have its own private blobs that cannot be decapsulated or encapsulated by any other user code, including any software running in the trusted mode.

When deploying the encrypted boot environment, the PRIBLOB setting must be advanced in the CAAM Security Configuration Register (SCFGR). The DEK blob must be initially generated with the default setting PRIBLOB=01 and the runtime software must set PRIBLOB=11. With PRIBLOB=11, the newly created blobs are not compatible with the blobs required to decrypt an encrypted boot image. When the HAB later executes the command to decrypt the DEK, an incompatible DEK blob is detected and causes

an error. A substitute encrypted boot image is neither decrypted nor executed. This ensures cryptographic separation of private blob types during the boot process and thereafter, avoiding any modification or replacement of DEK blobs.

NOTE

To enable the command used to set this bitfield in U-Boot, the U-Boot must be built with a custom KConfig. For more information, see [Appendix A.4, “Provisioning the PRIBLOB setting on the chip”](#).

NOTE

In the i.MX 7ULP, i.MX 7D, i.MX 6SoloX, and i.MX 6UL devices, the CAAM registers may be reset to suspend and resume operation. If deploying an encrypted boot environment using these devices, the PRIBLOB settings must be restored to ensure that the cryptographic separation of the private blob types is still valid.

The reference steps to set the PRIBLOB bitfield are described in [Appendix A.4, “Provisioning the PRIBLOB setting on the chip”](#).

5. References

- *i.MX 6 Reference Manual and Security Reference Manual*
- *i.MX 7 Reference Manual and Security Reference Manual*
- *HAB CST User Guide* available in the Code Signing Tool package downloadable from www.nxp.com (search for [CST_TOOL](#)).
- *Secure Boot on i.MX 50, i.MX 53, i.MX 6 and i.MX7 Series using HABv4* (document [AN4581](#)).
- *High-Assurance Boot Version 4 Application Programming Interface Reference Manual* available in the [Code Signing Tool package](#).

Appendix A. Development scripts

A.1. Building secure U-Boot

The following script builds the U-Boot with secure boot and encrypted boot support (including the dek_blob utility). This assumes that CONFIG_SECURE_BOOT is enabled for the MX6QSabreSD configuration at *include/configs/mx6sabre_common.h* (for older U-Boot versions—v2016.09 and lower) or at *configs/mx6qsabresd_defconfig* (for newer U-Boot versions—v2016.11 and higher).

To configure the U-Boot to be built with encrypted boot support, follow these steps (depending on the BSP release):

- **imx_v2016.03_04.1.15_2.0.0_ga and earlier:** define the following configurations in the board header file (*include/configs/mx6sabre_common.h*):
 - CONFIG_SYS_FSL_SEC_COMPAT 4 /* HAB version */.
 - CONFIG_FSL_CAAM.
 - CONFIG_CMD_DEKBLOB.

- CONFIG_SYS_FSL_LE.
- **imx_v2017.03_4.9.11_1.0.0_ga**: Select CONFIG_CMD_DEKBLOB in Kconfig/defconfig.
- **imx_v2017.03_4.9.88_2.0.0_ga and newer**: Select CONFIG_CMD_DEKBLOB in Kconfig/defconfig.

```
#!/bin/bash
uboot_dir="u-boot-imx location"
output_dir="$uboot_dir/output"
output_uboot="$output_dir/uboot"

export ARCH=arm
export CROSS_COMPILE=arm-linux-gnueabi-

target="mx6qsabresd_config"

cd $uboot_dir
if [[ ! -e $output_dir ]]
then
    mkdir $output_dir
fi
if [[ ! -e $output_uboot ]]
then
    mkdir $output_uboot
fi

make mrproper
make O=$output_uboot distclean
make O=$output_uboot $target
make O=$output_uboot
```

A.2. Building signed U-Boot

```
#!/bin/bash

uboot_dir="u-boot-imx location"
output_dir="$uboot_dir/output"
output_uboot="$output_dir/uboot"
output_signed="$output_dir/signed_uboot"
cst_dir="cst package location"
cst_signed_uboot="$cst_dir/sig_u-boot"

target_dir="$output_signed/boards/mx6q"

if [[ ! -d $target_dir ]]
then
    mkdir -p $target_dir
fi

if [[ ! -d $output_signed ]]
then
    mkdir $output_signed
fi

if [[ ! -d $cst_signed_uboot ]]
then
```

References

```
        mkdir $cst_signed_uboot
fi

cp $output_uboot/u-boot-dtb.imx $cst_signed_uboot
cd $cst_signed_uboot
#Get the u-boot image

#Create CSF file u-boot.csf as in Appendix B.1

#Signing u-boot image with CST
../linux64/bin/cst --o csf.bin --i u-boot.csf
#Padding CSF
objcopy -I binary -O binary --pad-to 0x2000 --gap-fill=0x00 csf.bin csf_pad.bin
#U-boot+Signature
cat u-boot-dtb.imx csf_pad.bin > u-boot-signed.bin
cp u-boot-signed.bin $target_dir
#Burn U-boot+signature to sd card
sudo dd if=u-boot-signed.bin of=/dev/sd<partition> bs=512 seek=2 && sync
```

A.3. Building encrypted U-Boot

The encrypted boot building process is divided into three steps.

1. Build the encrypted boot image and the DEK.
2. Provision the DEK to the target along with the signed boot image, generate the DEK blob, and copy it back to the host machine.
3. Construct the final encrypted boot image.

A.3.1. Encrypting and signing the U-Boot image

The process of first encrypting the U-Boot image and then signing it to ensure the integrity of the encrypted image with respect to the SRKs is achieved in two steps:

1. Encrypt the U-Boot image and generate the DEK.
2. Sign the encrypted U-Boot image, encrypt a dummy image, and generate a dummy DEK (to maintain the CSF commands—discard the dummy image and dummy DEK).

This script describes how to perform both steps:

```
#!/bin/bash

uboot_dir="u-boot-imx location"
output_dir="$uboot_dir/output"
output_uboot="$output_dir/uboot"
output_encrypted="$output_dir/encrypted_uboot"
cst_dir="cst package location"
cst_enc_uboot="$cst_dir/enc_u-boot"

target_dir="$output_encrypted/boards/mx6q"

if [[ ! -d $output_encrypted ]]
then
    mkdir $output_encrypted
fi
```

```

if [[ ! -d $target_dir ]]
then
    mkdir -p $target_dir
fi

if [[ ! -d $cst_enc_uboot ]]
then
    mkdir $cst_enc_uboot
fi

#Create a dummy image. Here u-boot is used as dummy image.
cp $output_uboot/u-boot-dtb.imx $cst_enc_uboot/u-boot-dtb.imx.dummy

#Get u-boot image for encryption
cp $output_uboot/u-boot-dtb.imx $cst_enc_uboot
cd $cst_enc_uboot

#Encrypt the u-boot image and generate DEK
../linux64/bin/cst --o csf_enc.bin --i u-boot_enc.csf

#Sign the encrypted u-boot image and encrypt a dummy image
../linux64/bin/cst --o csf_sign_enc.bin --i u-boot_sign_enc.csf

#Replace the Nonce/MAC generated by u-boot_sign_enc.csf with Nonce/MAC generated from u-
boot_enc.csf
NONCE_SIZE=12
read -p "MAC bytes used in CSF:" MAC_SIZE
case $MAC_SIZE in
"4"|"6"|"8"|"10"|"12"|"14"|"16")
echo "MAC bytes entered: "$MAC_SIZE;;
*)
echo "Incorrect MAC bytes or if not known please restart"
rm -rvf csf_sign_enc.bin csf_enc.bin
exit;;
esac

# Size of Nonce/MAC header and each size is made of 8 bytes
NONCEMAC_SIZE=`expr $NONCE_SIZE + $MAC_SIZE + 8`
NONCEMAC_OFFSET_IN_CSF=`expr $(stat -c%s "csf_enc.bin") - $NONCEMAC_SIZE`
dd if=csf_enc.bin of=noncemac.bin bs=1 skip=$NONCEMAC_OFFSET_IN_CSF count=$NONCEMAC_SIZE
echo "Replacing Nonce/MAC of second CSF with first CSF"
NONCEMAC_OFFSET_IN_CSF=`expr $(stat -c%s "csf_sign_enc.bin") - $NONCEMAC_SIZE`
dd if=noncemac.bin of=csf_sign_enc.bin bs=1 seek=$NONCEMAC_OFFSET_IN_CSF count=$NONCEMAC_SIZE

#Padded CSF
objcopy -I binary -O binary --pad-to 0x2000 --gap-fill=0xff csf_sign_enc.bin
csf_sign_enc_pad.bin
cp dek.bin $target_dir
cp u-boot-dtb.imx $target_dir
cp csf_sign_enc_pad.bin $target_dir
touch dek_blob.bin

```

A.3.2. Creating a DEK blob on the target

To create a DEK blob, the DEK created in [Appendix A.3.1, “Encrypting and signing the U-Boot image”](#) must be provisioned to the MX6QSabreSD board. Boot the device with the secure signed image generated before (*u-boot-signed.bin*). There are multiple ways to load the DEK to the target. Two of those methods are:

1. SD card and fatload method:
 - Copy *dek.bin* to the SD card partition:


```
cp dek.bin /media/<user>/Boot\ imx6qs/ && sync
```
 - On the target after booting a secure U-Boot image:


```
=> fatload mmc 1:1 0x17870000 dek.bin
```
2. TFTP method:
 - Copy *dek.bin* to the *tftboot* root directory:


```
cp dek.bin /<tftpboot_dir>/dek.bin
```
 - On the target:


```
=> tftp 0x17870000 dek.bin
```

When the data encryption key (*dek.bin*) is on the target, the *dek_blob* utility in the U-Boot can be used to create a DEK blob. The resulting blob information can be copied to the *dek_blob.bin* file created on the host machine.

- On the target:


```
=> dek_blob 0x17870000 0x17871000 128
```

Write the *dek_blob.bin* file to the SD card using the *fatwrite* command. It is enabled by selecting CONFIG_CMD_FAT in “Kconfig for u-boot BSP imx_v2018.03” (and higher). For older BSPs, add CONFIG_FAT_WRITE manually into the *BOARDNAME.h* file.

- On the target:


```
=> fatwrite mmc 1:1 0x17871000 dek_blob.bin 0x48
```

A.3.3. Assembling the encrypted boot image

This script assumes that the *dek_blob.bin* file contains the DEK blob generated with the signed U-Boot image.

```
#!/bin/bash
uboot_dir="u-boot-imx location"
output_dir="$uboot_dir/output"
output_uboot="$output_dir/uboot"
output_encrypted="$output_dir/encrypted_uboot"
cst_dir="cst package location"
cst_signed_uboot="$cst_dir/sig_u-boot"
cst_enc_uboot="$cst_dir/enc_u-boot"

target_dir="$output_encrypted/boards/mx6q"

if [[ ! -d $target_dir ]]
then
    mkdir -p $target_dir
fi
```



```

cp dek_blob.bin $target_dir
cd $cst_enc_uboot
#Concatenate encrypted u-boot, csf
cat u-boot-dtb.imx csf_sign_enc_pad.bin > u-boot_encrypted_no_dek.bin
#Pad binary
objcopy -I binary -O binary --pad-to 0x57c00 --gap-fill=0x00 u-boot_encrypted_no_dek.bin u-
boot_encrypted_no_dek_padded.bin
#Concatenate binary and dek
cat u-boot_encrypted_no_dek_padded.bin dek_blob.bin > u-boot_encrypted.bin

cp u-boot_encrypted.bin $target

#Download to SD card
sudo dd if=u-boot_encrypted.bin of=/dev/sd<partition> bs=512 seek=2 conv=fsync

```

A.4. Provisioning the PRIBLOB setting on the chip

To prevent the generation of DEK blobs usable at boot, a new U-Boot command is implemented to set the PRIBLOB bitfield of the SCFGR register of the CAAM IP to 0x3:

- *set_priblob_bitfield*—sets the PRIBLOB bitfield to 0x3 and displays its new value to check.

NOTE

When a bit is written, it cannot be cleared until the next reset. It means that if PRIBLOB is already set to 1 (b'01) and you try to write 2 (b'10), you get 3 (b'11).

To enable this command, check it in KConfig:

- ➔ Arm architecture ➔ Check “Support i.MX HAB features”.
- A new line appears: “Support the set_priblob_command”. It must be checked and the U-boot must be rebuilt.
- When the encrypted U-Boot is generated, it is recommended to set the PRIBLOB to 3. This way, the newly created blobs are incompatible to decrypt an encrypted boot image.

Appendix B. Troubleshooting

B.1. Signed U-Boot CSF example

```

[Header]
Version = 4.1
Hash Algorithm = sha256
Engine Configuration = 0
Certificate Format = X509
Signature Format = CMS
Engine = CAAM

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

```

References

```
[Install CSFK]
  File = "../crts/CSF1_1_sha256_2048_65537_v3_usr_cert.pem"

[Authenticate CSF]

[Install Key]
  Verification index = 0
  Target index = 2
  File = "../crts/IMG1_1_sha256_2048_65537_v3_usr_cert.pem"

[Authenticate Data]
  Verification index = 2
  Blocks = 0x177ff400 0x000 0x55c00 "u-boot-dtb.imx"
```

B.2. Encrypted U-Boot CSF example

CSF to encrypt a boot image and create a DEK (CSF-Enc):

```
[Header]
  Version = 4.1
  Hash Algorithm = sha256
  Engine Configuration = 0
  Certificate Format = X509
  Signature Format = CMS
  Engine = CAAM

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

[Install CSFK]
  File = "../crts/CSF1_1_sha256_2048_65537_v3_usr_cert.pem"

[Authenticate CSF]

[Install Key]
  Verification index = 0
  Target index = 2
  File = "../crts/IMG1_1_sha256_2048_65537_v3_usr_cert.pem"

[Authenticate Data]
  Verification index = 2
  Blocks = 0x177ff400 0x000 0xc00 "u-boot-dtb.imx"

[Install Secret Key]
  Verification index = 0
  Target index = 0
  Key = "dek.bin"
  Key Length = 192
  Blob address = 0x17857000

[Decrypt Data]
  Verification index = 0
  Mac Bytes = 16
  Blocks = 0x17800000 0xc00 0x55000 "u-boot-dtb.imx"
```

CSF to sign an encrypted image (CSF-Sign-Enc):

```
[Header]
  Version = 4.1
  Hash Algorithm = sha256
  Engine Configuration = 0
  Certificate Format = X509
  Signature Format = CMS
  Engine = CAAM

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

[Install CSFK]
  File = "../crts/CSF1_1_sha256_2048_65537_v3_usr_cert.pem"

[Authenticate CSF]

[Install Key]
  Verification index = 0
  Target index = 2
  File = "../crts/IMG1_1_sha256_2048_65537_v3_usr_cert.pem"

[Authenticate Data]
  Verification index = 2
  Blocks = 0x177ff400 0x000 0x55c00 "u-boot-dtb.imx"

[Install Secret Key]
  Verification index = 0
  Target index = 0
  Key = "dek.bin.dummy"
  Key Length = 192
  Blob address = 0x17857000

[Decrypt Data]
  Verification index = 0
  Mac Bytes = 16
  Blocks = 0x17800000 0xc00 0x55000 "u-boot-dtb.imx.dummy"
```

B.3. Script to determine the DEK blob address

```
#!/bin/bash
IVT_start=`hexdump -e '/4 "%X""\n"' -s 20 -n 4 u-boot-dtb.imx`
IVT_start_decimal=`echo $((16#$IVT_start))`
UBOOT_SIZE_decimal=`expr $(stat -c%s "u-boot-dtb.imx")`
#CSF padded size 0x2000
CSF_pad_size=8192
DEK_BLOB_addr_decimal=`expr $IVT_start_decimal + $UBOOT_SIZE_decimal + $CSF_pad_size`
echo "DEK blob address in CSF = IVT start address + SIZE of u-boot + Padded CSF (0x2000)"
DEK_BLOB_addr=`printf '%x\n' $DEK_BLOB_addr_decimal`
```

How to Reach Us:

Home Page:

www.nxp.com

Web Support:

www.nxp.com/support

Information in this document is provided solely to enable system and software implementers to use NXP products. There are no express or implied copyright licenses granted hereunder to design or fabricate any integrated circuits based on the information in this document. NXP reserves the right to make changes without further notice to any products herein.

NXP makes no warranty, representation, or guarantee regarding the suitability of its products for any particular purpose, nor does NXP assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation consequential or incidental damages. "Typical" parameters that may be provided in NXP data sheets and/or specifications can and do vary in different applications, and actual performance may vary over time. All operating parameters, including "typicals," must be validated for each customer application by customer's technical experts. NXP does not convey any license under its patent rights nor the rights of others. NXP sells products pursuant to standard terms and conditions of sale, which can be found at the following address:

www.nxp.com/SalesTermsandConditions.

While NXP has implemented advanced security features, all products may be subject to unidentified vulnerabilities. Customers are responsible for the design and operation of their applications and products to reduce the effect of these vulnerabilities on customer's applications and products, and NXP accepts no liability for any vulnerability that is discovered. Customers should implement appropriate design and operating safeguards to minimize the risks associated with their applications and products

NXP, the NXP logo, NXP SECURE CONNECTIONS FOR A SMARTER WORLD, COOLFLUX, EMBRACE, GREENCHIP, HITAG, I2C BUS, ICODE, JCOP, LIFE VIBES, MIFARE, MIFARE CLASSIC, MIFARE DESFire, MIFARE PLUS, MIFARE FLEX, MANTIS, MIFARE ULTRALIGHT, MIFARE4MOBILE, MIGLO, NTAG, ROADLINK, SMARTLX, SMARTMX, STARPLUG, TOPFET, TRENCHMOS, UCODE, Freescale, the Freescale logo, Altivec, C-5, CodeTEST, CodeWarrior, ColdFire, ColdFire+, C-Ware, the Energy Efficient Solutions logo, Kinetis, Layerscape, MagniV, mobileGT, PEG, PowerQUICC, Processor Expert, QorIQ, QorIQ Qonverge, Ready Play, SafeAssure, the SafeAssure logo, StarCore, Symphony, VortiQa, Vybrid, Airfast, BeeKit, BeeStack, CoreNet, Flexis, MXC, Platform in a Package, QUICC Engine, SMARTMOS, Tower, TurboLink, and UMEMS are trademarks of NXP B.V. All other product or service names are the property of their respective owners. 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 or registered trademarks of Arm Limited (or its subsidiaries) 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. Oracle and Java are registered trademarks of Oracle and/or its affiliates. The Power Architecture and Power.org word marks and the Power and Power.org logos and related marks are trademarks and service marks licensed by Power.org.

© 2018 NXP B.V.

Document Number: AN12056

Rev. 0

11/2018

