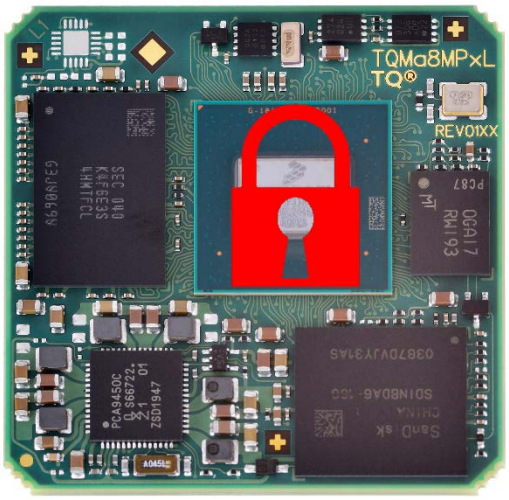
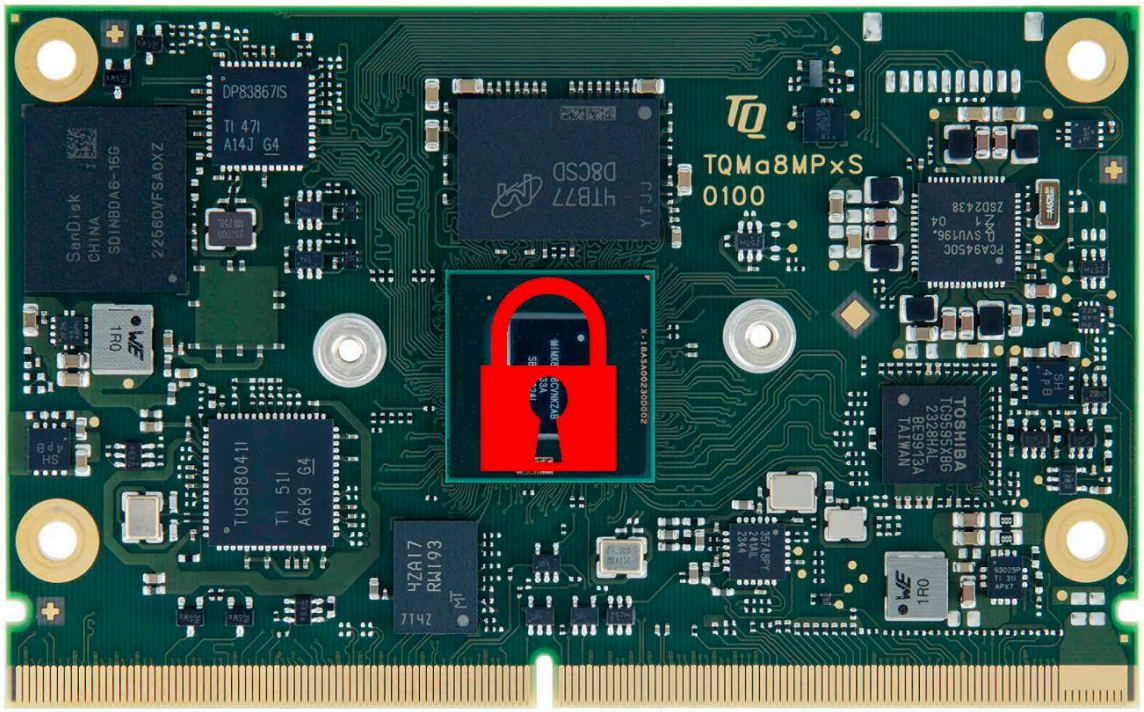
How-to:

Secure Boot TQMa8MPxL/S

26.02.2026





The host PC used in this guide uses Linux (Ubuntu 22.04) as operating system.

	**ATTENTION:** Fuses (One Time Programmable) are set in this How-to, this process is irreversible. It is therefore strongly recommended to use a development pattern for this guide.

1. Procedure

This guide explains how a chain of trust can be established from the boot loader via the Linux kernel to a root partition with dm-verity.

The following table provides a simplified description of the steps involved in creating the chain of trust and verification during the boot process:

Creation	Execution
U-Boot	
The bootloader U-Boot is signed with the private key of an asymmetric key pair. The signature and the public key are integrated into the U-Boot image. A hash of the public key is written to the fuses of the SoC.	The Boot ROM loads the U-Boot image and extracts the signature block with the signature and the public key. Using these two components and the hash of the public key from the fuses, the U-Boot image can be verified.
FIT-Image	
The hash of the FIT image is signed with the private key of an asymmetric key pair. This signature is attached to the FIT image. The public key is written to the U-Boot devicetree.	U-Boot verifies the FIT image during loading using the signature and the public key.
Root file system	
Creation of a root partition with veritysetup. This creates a root hash of all files in the file system, which must be stored in Initramfs.	Creation of a device mapper from the root partition by specifying the root hash in Initramfs. Mount of the device mapper as a root file system. Specifying the root hash guarantees that the files in the root partition are unchanged.

2. Preparation

The following projects are required to create a signed boot stream for TQMa91xx:

- imx-mkimage: https://github.com/nxp-imx/imx-mkimage (required)
- NXP Code Signing Tool 3.4.x (with NXP Account): https://www.nxp.com/webapp/Download?colCode=IMX_CST_TOOL_NEW (required)
- TQ Yocto-Workspace: https://github.com/tq-systems/ci-meta-tq (recommended)

The bootstream for TQMa8MPxL/S consists of several artifacts. To obtain all these artifacts from the same source, it is recommended to use the TQ Yocto workspace ci-meta-tq. The instructions included there can be followed to build a complete image (`tq-image-weston-debug` or `tq-image-generic-debug`) for one of the following TQMa8MPxL/S-based devices:

- tqma8mpxl-mba8mpxl.conf
- tqma8mpxs-mb-smarc-2

ATTENTION: To create U-Boot with secure boot functionality (HAB4), the following line must be added to `local.conf`:

```
DISTRO_FEATURES:append = " secure"
```

Next, the boot stream must be recreated:

```
$ bitbake imx-boot
```

The TQ Yocto workspace can also be used to create an image of the complete chain of trust presented here. The settings required for this are described in section 5.2 .

The sources for the Linux kernel and U-Boot are optional but recommended. They can be downloaded from Github:

Linux: https://github.com/tq-systems/linux-tqmaxx/tree/TQM-linux-v6.12.y
U-Boot: https://github.com/tq-systems/u-boot-tqmaxx/tree/TQM-lf_v2024.04

Linux and U-Boot should already be compiled for a variant of TQMa8MPxL/S in preparation.

3. U-Boot

3.1 Generating keys

Signing and verification of the boot stream are carried out using a public key infrastructure (PKI). If not already available, the Code Signing Tool can be used to create a suitable PKI. The CST 3.4.x is a tar.gz archive that only needs to be unpacked. No further installation is necessary. The following steps can be used to generate the sample keys for this guide:

	**ATTENTION**: Paths are relative to the folder extracted from the archive.

1. Enter the serial number of the first certificate in `keys/serial` (file must be created):

```
12345678
```

2. Enter the passphrase twice in `keys/key_pass.txt` (file must be created):

```
my_passphrase
my_passphrase
```

3. Create PKI tree:

```
$ keys/hab4_pki_tree.sh -existing-ca n -kt ecc -kl p521 -duration 10 –num-srk 4
-srk-ca y
```

For an explanation of the options, please refer to the User Guide contained in the CST (in the `docs` subfolder) or the `--help` option of the above script.

Alternatively, the script can also be called without options and configured in interactive mode.

The script generates keys in `keys/` and certificates in `crts/`.

4. Create SRK table and SRK hash table:

```
$ linux64/bin/srktool –h 4 -d sha256 -t SRK_1_2_3_4_table.bin \
-e SRK_1_2_3_4_fuse.bin -f 1 -c
crts/SRK1_sha256_secp521r1_v3_ca_crt.pem,crts/SRK2_sha256_secp521r1_v3_ca_crt.pe
m,crts/SRK3_sha256_secp521r1_v3_ca_crt.pem,crts/SRK4_sha256_secp521r1_v3_ca_crt.
pem
```

5. Write SRK hash table in fuses:

	**ATTENTION**: This step is only possible once and is irreversible. The following values are only examples and must be replaced by your own values.

a. Display hashes:

```
$ hexdump -e '/4 "0x"' -e '/4 "%X""\n"' SRK_1_2_3_4_fuse.bin
0x00000000
0x11111111
0x22222222
0x33333333
0x44444444
0x55555555
0x66666666
0x77777777
```

b. Write hashes in fuses (TQMa8MPxL/S U-Boot):

```
=> fuse prog 16 0 0x00000000
=> fuse prog 16 1 0x11111111
=> fuse prog 16 2 0x22222222
=> fuse prog 16 3 0x33333333
=> fuse prog 16 4 0x44444444
=> fuse prog 16 5 0x55555555
=> fuse prog 16 6 0x66666666
=> fuse prog 16 7 0x77777777
```

3.2 Creating a signed boot stream

Note: The device tree binary for U-Boot is required for this step. Ready-made device tree binaries can be found in the Yocto workspace in the U-Boot build directory. The path to the build directory can be displayed with `bitbake virtual/bootloader -e | grep ^B=`.

1. Copy the required files (successful build of a TQ image, see above, or the U-Boot sources is assumed):
   a. ARM Trusted Firmware: `${DEPLOY_DIR_IMAGE}/bl31-imx8mp.bin`, rename to `bl31.bin`
   b. U-Boot SPL:
      `${DEPLOY_DIR_IMAGE}/u-boot.bin`
      This file is a link, so copy it with `cp --dereference` or display and copy the original file with `ls --long`
      or
      from self-compiled U-Boot sources
   c. U-Boot Proper:
      `${DEPLOY_DIR_IMAGE}/u-boot.bin`
      This file is a link, so copy it with `cp --dereference` or display and copy the original file with `ls --long`
      or
      from self-compiled U-Boot sources

d. RAM Firmware: `${DEPLOY_DIR_IMAGE}/lpddr4*.bin`
   The version in the file name must be removed by renaming it, e.g. `lpddr4_pmu_train_1d_dmem_202006.bin` to `lpddr4_pmu_train_1d_dmem.bin`

e. U-Boot Proper without DTB:
   e.g. `<U-Boot BuildFolder>/u-boot-nodtb.bin`

f. U-Boot DTB
   e.g. `<U-Boot BuildFolder>/arch/arm/dts/imx8mp-tqma8mpql-mba8mpxl.dtb`

g. U-Boot mkimage Tool
   `<U-Boot BuildFolder>/tools/mkimage`, umbenennen in `mkimage_uboot`

These files must be copied to `imx-mkimage/iMX8M/`. `imx-mkimage` can be obtained from the Github repository mentioned above, no installation is necessary.

2. Build Bootstream (in folder `imx-mkimage`):

```
$ make -j8 SOC=iMX8MP REV=A0 dtbs=imx8mp-tqma8mpql-mba8mpxl.dtb
flash_spl_uboot
…
========= OFFSET dump =========
Loader IMAGE:
 header_image_off      0x0
 dcd_off          0x0
 image_off        0x40
 csf_off          0x3cc00
 spl hab block:   0x91ffc0 0x0 0x3cc00

Second Loader IMAGE:
 sld_header_off   0x58000
 sld_csf_off            0x59020
 sld hab block:   0x401fadc0 0x58000 0x1020
 fit-fdt csf_off  0x5b020
 fit-fdt hab block:     0x401fadc0 0x58000 0x3020
SPL CSF block:
      Blocks =    0x91ffc0 0x0 0x3cc00 "flash.bin"
SLD CSF block:
      Blocks =    0x401fadc0 0x58000 0x1020 "flash.bin",\
SLD FIT-FDT CSF block:
      Blocks =    0x401fadc0 0x58000 0x3020 "flash.bin"
```

ATTENTION: The offsets for the container and signature block are required in the next step.

3. Output the address, offset, and length of relevant blocks:

```
$ make -j8 SOC=iMX8MP REV=A0 dtbs=imx8mp-tqma8mpql-mba8mpxl.dtb print_fit_hab
…
0x40200000 0x5D000 0xFA6C8
0x402FA6C8 0x1576C8 0xA168
0x970000 0x161830 0xAA70
```

Next, copy the artifact `imx-mkimage/iMX8M/flash.bin` to the CST folder extracted in step “3.1 Generating keys.”

4. Create Command Sequence Files (CSF):

   CSF based on: https://github.com/nxp-imx/uboot-imx/blob/lf_v2024.04/doc/imx/habv4/csf_examples/mx8m/csf_spl.txt and https://github.com/nxp-imx/uboot-imx/blob/lf_v2024.04/doc/imx/habv4/csf_examples/mx8m/csf_fit.txt

   The files are stored in the CST folder extracted in step “3.1 Generating keys” with the following names and contents.

   a. csf_spl.txt: Enter the values for `spl hab block` from step 2 “Build bootstream” under `[Authenticate Data]`

```
[Header]
    Version = 4.3
    Hash Algorithm = sha256
    Engine = CAAM
    Engine Configuration = 0
    Certificate Format = X509
    Signature Format = CMS

[Install SRK]
    # Index of the key location in the SRK table to be installed
    File = "SRK_1_2_3_4_table.bin"
    Source index = 0

[Install CSFK]
    # Key used to authenticate the CSF data
    File = "crts/CSF1_1_sha256_secp521r1_v3_usr_crt.pem"

[Authenticate CSF]

[Unlock]
    # Leave Job Ring and DECO master ID registers Unlocked
    Engine = CAAM
    Features = MID

[Install Key]
    # Key slot index used to authenticate the key to be installed
    Verification index = 0
    # Target key slot in HAB key store where key will be installed
    Target index = 2
    # Key to install
    File = "crts/IMG1_1_sha256_secp521r1_v3_usr_crt.pem"

[Authenticate Data]
    # Key slot index used to authenticate the image data
    Verification index = 2
    # Authenticate Start Address, Offset, Length and file
    Blocks = 0x91ffc0 0x0 0x3cc00 "flash.bin"
```

b. csf_fit.txt: Enter the values for `SLD CSF block` from step 2 “Build Bootstream” and the values from step 3 “Output the address, offset, and length of relevant blocks” under `[Authenticate Data]`

```
[Header]
    Version = 4.3
    Hash Algorithm = sha256
    Engine = CAAM
    Engine Configuration = 0
    Certificate Format = X509
    Signature Format = CMS

[Install SRK]
    # Index of the key location in the SRK table to be installed
    File = "SRK_1_2_3_4_table.bin"
    Source index = 0

[Install CSFK]
    # Key used to authenticate the CSF data
    File = "crts/CSF1_1_sha256_secp521r1_v3_usr_crt.pem"

[Authenticate CSF]

[Install Key]
    # Key slot index used to authenticate the key to be installed
    Verification index = 0
    # Target key slot in HAB key store where key will be installed
    Target index = 2
    # Key to install
    File = "crts/IMG1_1_sha256_secp521r1_v3_usr_crt.pem"

[Authenticate Data]
    # Key slot index used to authenticate the image data
    Verification index = 2
    # Authenticate Start Address, Offset, Length and file
    Blocks = 0x401fadc0 0x58000 0x1020 "flash.bin", \
             0x40200000 0x5D000 0xFA6C8 "flash.bin", \
             0x402FA6C8 0x1576C8 0xA168 "flash.bin", \
             0x970000 0x161830 0xAA70 "flash.bin"
```

5. Sign Bootstream

```
linux64/bin/cst -i csf_spl.txt -o csf_spl.bin
```

and

```
linux64/bin/cst -i csf_fit.txt -o csf_fit.bin
```

The signed individual parts must then be inserted into the boot stream:

```
dd if=csf_spl.bin of=flash.bin seek=$((0x3cc00)) bs=1 conv=notrunc
```

and

```
dd if=csf_spl.bin of=flash.bin seek=$((0x59020)) bs=1 conv=notrunc
```

The appropriate offsets are output in step 2, “Build Bootstream”:
For csf_spl.bin:

```
csf_off           0x3cc00
```

For csf_fit.bin:

```
sld_csf_off             0x59020
```

The steps for replacing the boot stream can be found in the BSP layer (https://github.com/tq-systems/meta-tq) under meta-tq/doc.

3.3 Verification

To check if the signed boot stream is valid, use the `hab_status` command in U-Boot:

```
=> hab_status

Secure boot disabled

HAB Configuration: 0xf0, HAB State: 0x66
No HAB Events Found!
```

If an event is found, the boot stream is invalid and would not be able to boot on a locked device.

For falsification, an unsigned bootstream can be booted and then `hab_status` can be called:

```
=> hab_status

Secure boot disabled

HAB Configuration: 0xf0, HAB State: 0x66

--------- HAB Event 1 -----------------
event data:
     0xdb 0x00 0x08 0x45 0x33 0x11 0xcf 0x00

STS = HAB_FAILURE (0x33)
RSN = HAB_INV_CSF (0x11)
CTX = HAB_CTX_CSF (0xCF)
ENG = HAB_ENG_ANY (0x00)

--------- HAB Event 2 -----------------
event data:
     0xdb 0x00 0x14 0x45 0x33 0x0c 0xa0 0x00
     0x00 0x00 0x00 0x00 0x00 0x91 0xff 0xc0
     0x00 0x00 0x00 0x20

STS = HAB_FAILURE (0x33)
RSN = HAB_INV_ASSERTION (0x0C)
CTX = HAB_CTX_ASSERT (0xA0)
ENG = HAB_ENG_ANY (0x00)

--------- HAB Event 3 -----------------
event data:
```

```
     0xdb 0x00 0x14 0x45 0x33 0x0c 0xa0 0x00
     0x00 0x00 0x00 0x00 0x00 0x91 0xff 0xe0
     0x00 0x00 0x00 0x0c

STS = HAB_FAILURE (0x33)
RSN = HAB_INV_ASSERTION (0x0C)
CTX = HAB_CTX_ASSERT (0xA0)
ENG = HAB_ENG_ANY (0x00)

--------- HAB Event 4 -----------------
event data:
     0xdb 0x00 0x14 0x45 0x33 0x0c 0xa0 0x00
     0x00 0x00 0x00 0x00 0x00 0x92 0x00 0x00
     0x00 0x00 0x00 0x04

STS = HAB_FAILURE (0x33)
RSN = HAB_INV_ASSERTION (0x0C)
CTX = HAB_CTX_ASSERT (0xA0)
ENG = HAB_ENG_ANY (0x00)
```

3.4 Lock the device

	**ATTENTION:** This step is irreversible and should only be carried out if necessary. If the configuration is incorrect, this step will result in an unusable device.

The device can be locked by setting a fuse. This ensures that only valid Bootstreams verified by the boot ROM are booted:

```
=> fuse prog 1 3 0x2000000
```

4. FIT-Image

ATTENTION: Path information is relative to a new, empty folder, e.g. `fit_image_work`, or the kernel sources, if self-compiled. Hereafter referred to as the working directory.

4.1 Generating a key pair

An asymmetric key pair is used to sign the FIT image. Such a pair can be generated with OpenSSL:

```
$ openssl genpkey -algorithm RSA -out dev.key -pkeyopt rsa_keygen_bits:2048
$ openssl req -batch -new -x509 -key dev.key -out dev.crt
```

4.2 Create image tree source

Create image tree source `sign.its` for the FIT image.

```
/dts-v1/;

/ {
  description = "Kernel fitImage for TQMa91xx";
  #address-cells = <1>;

  images {
    kernel-1 {
      description = "Linux kernel";
      data = /incbin/("Image");
      type = "kernel";
      arch = "arm64";
      os = "linux";
      compression = "gzip";
      load = <0x90000000>;
      entry = <0x90000000>;
      hash-1 {
        algo = "sha256";
      };
    };

    fdt-1 {
      description = "Flattened Device Tree blob";
      data = /incbin/("<path/to/Devicetree.dtb>");
      type = "flat_dt";
      arch = "arm64";
      compression = "none";
      load = <0x97000000>;
      hash-1 {
        algo = "sha256";
      };
```

```
    };
  };

  configurations {
    default = "conf-1";
    conf-1 {
      description = "Linux kernel, FDT blob";

      kernel = "kernel-1";
      fdt = "fdt-1";

      hash-1 {
        algo = "sha256";
      };

      signature-1 {
        algo = "sha256,rsa2048";
        key-name-hint = "dev";
        padding = "pkcs-1.5";
        sign-images = "kernel", "fdt";
      };
    };
  };
};
```

4.3 Creating a signed FIT image

Note: The devicetree binary for U-Boot is required for this step. Ready-made devicetree binaries can be found in the Yocto workspace in the U-Boot build directory. The path to the build directory can be displayed with `bitbake virtual/bootloader –e | grep ^B=`.

1. Copy the required files into the working directory:
    a. Rename U-Boot devicetree `imx8mp-tqma8mpql-mba8mpxl.dtb` or `imx8mp-tqma8mpxs-mb-smarc-2.dtb`, in `pubkey.dtb`:

       From U-Boot build directory in Yocto workspace (path: `bitbake virtual/bootloader –e | grep ^B=`)

       or

       from self-compiled U-Boot sources

b. Linux-Kernel:

`${DEPLOY_DIR_IMAGE}/Image`

This file is a link, so copy it with `cp --dereference` or display and copy the original file with `ls --long`

or

from self-compiled Linux sources

c. Linux devicetree:

Depending on the variant `${DEPLOY_DIR_IMAGE}/imx8mp-tqma8mp…`

This file is a link, so copy it with `cp --dereference` or display and copy the original file with `ls --long`

or

from self-compiled Linux sources

d. The keys generated in step 4.1

e. The ITS file generated in step 4.2

2. Create FIT image with signature

```
$ mkimage -f sign.its -K pubkey.dtb -k . -r image.itb
```

The public key is written to the devicetree of the U-Boot. This key is used to verify the FIT image signed above.

ATTENTION: To pack the U-Boot devicetree with the public key into the signed bootstream from chapter 3.2, the steps from chapter 3.2 must be repeated with a customized U-Boot Proper `u-boot.bin`. To do this, the devcietree with the public key `pubkey.dtb` must be specified via the `EXT_DTB` option when compiling the U-Boot:

`make EXT_DTB=<Pfad/zu/pubkey.dtb>`

4.4 Verification

In U-Boot with public keys, the signed FIT image `image.itb` can be booted with `bootm` after it has been loaded from a suitable medium (TFTP, eMMC, SD).

When booting the FIT image, U-Boot returns the information `Verifying Hash Integrity ... sha256,rsa2048:dev+ OK` with name, algorithm and length of the key generated in chapter 4.1 on the console:

```
## Loading kernel from FIT Image at 80400000 ...
   ...
   Verifying Hash Integrity ... sha256,rsa2048:dev+ OK
   ...
## Loading ramdisk from FIT Image at 80400000 ...
   ...
   Verifying Hash Integrity ... sha256,rsa2048:dev+ OK
   ...
## Loading fdt from FIT Image at 80400000 ...
```

```
   ...
   Verifying Hash Integrity ... sha256,rsa2048:dev+ OK
   ...
```

For falsification, another key pair can be generated as described in chapter 4.1 and used to sign the FIT image. This FIT image cannot be booted without exchanging the key in the U-Boot Devicetree:

```
## Loading kernel from FIT Image at 80400000 ...
   Using 'conf-1' configuration
   Verifying Hash Integrity ... sha256,rsa2048:test-  error!
Verification failed for '<NULL>' hash node in 'conf-1' config node
Failed to verify required signature 'key-dev'
Bad Data Hash
ERROR: can't get kernel image!
```

5. Extend Chain of Trust: root partition

The previously established chain of trust verifies the origin of the U-Boot and Linux kernel. With the mechanisms mentioned above, only the owner of the generated private key can sign his software and boot it on the device. Further links can be added to the chain. The following section outlines how the root partition can be protected against manipulation using dm-verity. For the real implementation, it is also shown how the complete chain can be created with the TQ-BSP. A step-by-step guide to dm-verity protection is omitted due to the complexity of the requirements.

5.1 Sketch: Verity Devicemapper

1. Generate Verity hashes:

   `veritysetup` calculates the hash values and stores them at the end of the root partition. The root partition can be a real file or a block device file (e.g. `/dev/sdaX`).

   ```
   veritysetup \
     --data-block-size=1024 \
     --hash-block-size=4096 \
     --hash-offset=<Offset> \
     format \
     <Root-Partition.img> \
     <Root-Partition.img>
   ```

   `veritysetup` outputs the following information (with correspondingly different values):

   ```
   VERITY header information for data.img
   UUID:                   e06ff4cb-6b56-4ad4-bd97-0104505a70a5
   Hash type:              1
   Data blocks:            204800
   Data block size:        1024
   Hash block size:        4096
   Hash algorithm:         sha256
   Salt:      17328c48990b76fbb3e05d0ebfd236043674cf0d14c278bc875b42693621cc21
   Root hash: a0e1a449d452f74d041706b955794c0041e3d8ad051068df6589e08485323698
   ```

   The root hash is the sensitive value that needs to be protected. If this hash is compromised, e.g. if it can be changed by an unauthorized person, then the protection of the integrity of the root partition by dm-verity is worthless.

2. Integrate the root hash into the chain of trust

   The root hash generated above is stored in the signed FIT image, which protects it against manipulation. For this purpose, an initramfs is added to the FIT image in which the root hash is stored in a file.

   The `images` node of the ITS file from chapter 4.2 is extended by the following section, among others:

   ```
   ramdisk-1 {
     description = "dm-verity-image-initramfs";
     data = /incbin/("<path/to/Initramfs.cpio.gz>");
     type = "ramdisk";
     arch = "arm64";
   ```

```
  os = "linux";
  compression = "none";
  load = <0x98000000>;
  entry = <0x98000000>;
  hash-1 {
    algo = "sha256";
  };
};
```

3. Check the integrity of the root partition

   The initramfs contains a suitable script that generates a device mapper from the root partition and the root hash.

```
veritysetup \
  --data-block-size=${DATA_BLOCK_SIZE} \
  --hash-offset=${DATA_SIZE} \
  create rootfs \
  </dev/Root-Paritition> \
  </dev/Root-Paritition> \
  <Root Hash>
```

   The device mapper is then mounted:

```
mount \
  -o ro \
  /dev/mapper/rootfs \
  /rootfs
```

   The root filesystem is read-only. To switch to the actual root filesystem, use `switch-root`.

5.2 Automated creation with TQ-BSP

In principle, an image with a chain of trust from the boot loader to the root partition can be created automatically with the TQ-BSP.

For TQMa8MPxL/S the following options have to be added to `local.conf` :

```
# The DISTRO_FEATURE secure necessary config options for U-Boot and Kernel
DISTRO_FEATURES:append = " secure"
# Name of the key used for signing the bootloader
IMX_HAB_KEY_NAME = "hab4"
# Activates the signing of the FIT image in the build process
UBOOT_SIGN_ENABLE = "1"
# This class contains the logic for creating a protected root partition
IMAGE_CLASSES += "dm-verity-img"
# Name of the initramfs image for dm-verity handling
INITRAMFS_IMAGE = "dm-verity-image-initramfs"
# Initramfs is stored as a separate artifact in the image
INITRAMFS_IMAGE_BUNDLE = "0"
# Store FIT image with initramfs in boot partition
```

```
IMAGE_BOOT_FILES:append = " fitImage-${INITRAMFS_IMAGE}-${MACHINE}-
${MACHINE};fitImage"
# Image to be protected with dm-verity
# Alternative: tq-image-weston-debug
DM_VERITY_IMAGE = "tq-image-generic-debug"
# Type of the above image
DM_VERITY_IMAGE_TYPE = "ext4"
```

ATTENTION: The exact options may change in future versions of the BSP. The latest information can be found in the BSP layer documentation (https://github.com/tq-systems/meta-tq) under meta-tq/doc.

The complete image is created with `bitbake tq-image-generic-debug` and can then be written to an SD card, for example.

5.3 Verification

In Linux, `mount -a` can be used to check if the Verity Devicemapper is mounted as the root filesystem:

```
# mount
...
/dev/mapper/rootfs on / type ext4 (ro,relatime)
...
```

In addition, the entire root file system is read-only in this case:

```
# touch test
touch: cannot touch 'test': Read-only file system
```

For falsification, the root file system can be modified offline and the device rebooted. The modification causes a different root hash and the boot process is aborted:

```
device-mapper: verity: 179:98: data block 1 is corrupted
```

More information about the TQMa8MPxL/S can be found in the TQ Support Wiki: https://support.tq-group.com/en/arm/modules#nxp_imx_8m_series

TQ-Systems GmbH
Mühlstraße 2 I Gut Delling I 82229 Seefeld
Info@TQ-Group | TQ-Group