Apertis secure boot

Contents

For both privacy and security reasons it is important for modern devices to ensure that the software running on the device hasn't been tampered with. In particular any tampering with software early in the boot sequence will be hard to detect later while having a big amount of control over the system. To solve this issues various vendors and consortiums have created technologies to combat this, known under names as "secure boot", "highly assured boot"(NXP), "verified boot"(Google Android/ChromeOS).

While the scope and implementation details of these technologies differs the approach to provide a trusted boot chain tends to be similar between all of them. This document discusses how that aspect of the various technologies works on a high-level and how this can be introduced into Apertis.

Boot sequence

To understand how secure boot works first one has to understand how booting works. From a high-level perspective a CPU is a very simple beast, it needs to be pointed at a stream of instructions (code) which it will then be able to execute. Without instructions a CPU cannot do anything. The instructions also need to be in a region of memory which the CPU can access. However when a device is powered on the code that is meant to be run on it (e.g. Linux) will not be in memory yet. To make matters worse on power on main memory (Dynamic RAM) will not even be accessible by the CPU yet! To solve this problem some bootstrapping is required, typically referred to as booting the system.

The very first step in the boot process after power on is to get the CPU to start executing some instructions. As the CPU cannot load instructions without running instructions these first instructions are hardwired into the SoC directly with the CPU is hardwired to start executing those when powers comes on. This hardwired piece of code is often referred to as the ROM or romcode.

The job of the romcode is to do very basic SoC setup and load further code to execute. To allow the romcode to do its job, it will have access to a small amount of static RAM (SRAM, typically 64 to 128 kilobyte). The locations from where the ROM code can load is system specific. On most modern ARM-based systems this will include at least (SPI-connected) flash (NAND/NOR), eMMC cards, SD cards, serial ports etc. Most systems can only have code loaded over USB initially while some can even load code directly over the network via bootp!. The details of the format the code needs to be in (e.g. specific headers), how the code is presented (e.g. specific offsets on the eMMC) is very system specific. Once romcode managed to load the code from one of its supported location into SRAM execution of that code will start, which will the first time user supplied code is actually ran on the device.

This next step is known under various different names such as Boot Loader stage 1(BL1), Secondary Program Loader(SPL), Tertiary Program Loader(TPL), etc. The code for this stage must be quite small as only SRAM is available at this stage. The goal for this step is normally to initialize Dynamic RAM (e.g. run DDR memory training) followed by loading the next step into DRAM and executing it (which can be far bigger now that DRAM is available). Depending on the system this stage may also provide initial user feedback that the system is booting (e.g. display a first splash image, turning an LED on etc), but that purely depends on the overall system design and available space.

What the next step of executed code is more system specific. In some cases it can directly be Linux, in some cases it will be a bootloader with more functionality (as all of main memory is now available) and in some cases it will be multiple loader steps. As an example of the last case for devices using ARM Trusted Firmware there will typically be follow-on steps to load the secure firmware (such as OP-TEE[1]) followed by a non-secure world bootloader which loads Linux. For those interested the various images used in an ATF setup can be found here[2].

Linux starting up typically is the last phase of the boot process. For Linux to start the previous stage will have loaded a kernel image, optionally a initramfs and optionally a devicetree into main memory. The combination of these will load the root filesystem at which point userspace (e.g. applications) will start running.

Note that while the above is a simple view on the basic boot process, the overall flow will be the same on all systems (both ARM and non-ARM devices). For the above we also implicitly assumed that only one CPU is booted, for some more complex systems multiple CPUs (e.g. main application processors and various co-processors) might be booted. It may even be the case that all the early stages are done by a co-processor which takes care of loading the first code and starting the main processor. The overall description is also valid for system with hypervisors, essentially the hypervisor is just another stage in the boot sequence and will load/start the code for each of the cells it runs.

For this document we'll only look at securing the booting of the main (Linux running) processor without a hypervisor.

Secure boot sequence

The main objective for a secure boot process is to ensure all code that gets executed by the processor is trusted. As each of the stages described in the previous section is responsible for loading the code for the next stage the solution for that is relatively straight-forward. Apart from loading the next stage of code, each stage also needs to verify the code it has loaded. Typically this is done by some signature verification mechanism.

The ROM step is normally assumed to be fully trusted as it's hard-wired into the SoC and cannot be replaced. How the ROM is configured and how it validates the next stage is highly device specific. Later steps can do the verification either by calling back into ROM code (thus re-using the same mechanisms as the ROM) or by pure software implementation (making it more consistent between different devices).

In all cases to support this, apart from device specific configuration, all boot stages need to be appropriately signed. Luckily this is typically based on standard mechanisms such as RSA keys and X.509 Certificates.

Once Linux starts the approach has to be different as it's not feasible in most systems to fully verify all of the root filesystem at boot time as this simply

[1] https://www.apertis.org/concepts/op-tee/

[2] https://trustedfirmware-a.readthedocs.io/en/latest/getting_started/image-terminology.html

would take far too long. As such the form of protection described thus far only gets applied up to the point the Linux kernel starts loading.

Threat models

To understand what a secure boot system really secures it's important to look at the related threat models. As a first step we can distinguish between offline (device is turned off) and online attacks (device powered on).

For these considerations the assumption is made all boot steps work as intended. As with any software security vulnerabilities can invalidate the protection given. While in most cases these can be patches as issues become known, for ROM code this is impossible without a hardware change.

offline attacks

- Attack: Replace any of the boot stages on device storage (physical access required)
- Impact: Depending on the boot stage the attacker can get full control of the device for each following boot.
- Mitigation: Assuming each stage correctly validates the next boot stage, any tampering with loaded code will be detected and prevented (e.g. device fails to boot).
- Attack: Trigger the device to load software from external means (e.g. USB or serial) under the attackers control.
- Impact: Depending on the boot stage the attacker can get full control of the device.
- Mitigation: The ROM or any stage that loads from an external source should use the same verification as for any on device stages. However for production use, if possible, loading software from external source should be disabled.
- Attack: Replace or add binaries on the systems root filesystem
- Impact: Full control of the device as far as the kernel allows.
- Mitigation: No protection from the above mechanisms.

online attacks

- Attack: Gain enough access to replace any of the boot stages on device storage
- Impact: Depending on the boot stage the attacker can get full control of the device for each following boot.

- Mitigation: Assuming each stage correctly validates the next boot stage, any tampering with loaded code will be detected and prevented (e.g. device fails to boot).
- Attack: Replace or add binaries on the systems root filesystem
- Impact: Full control of the device as far as the kernel allows.
- Mitigation: No protection from the above mechanisms.

Signing and signing infrastructure

To securely boot a device it is assumed all the various boot stages have some kind of signature which can be validate by previous stages. Which by extension also means the protection is only as strong as the signature; if an attacker can sign code under their control with a signature that is valid (or seen as valid) for the verifying step all protection is lost. This means that special care has to be taken with respect to key handling to ensure signing keys are kept with the right amount of security depending on their intended use.

For development usage and devices a low amount of security is ok in most cases, the intention in the development stage is for developers to be easily able to run their own code and by extension should be able to sign their own builds with minimal effort.

For production devices however the requirements should be much more strict as unauthorized of control of a signing key can allow attackers to defeat the intended protection by secure boot. Furthermore production devices should typically not be allowed to run development builds as those tend to enable extra access for debugging and development reasons which tend to be a great attack vector.

For these reason it's recommendable to have at least two different sets of signing keys, one for development usage and one for production use. Development keys can be kept with low security or even be publicly available, while production keys should *only* be used to sign final production images and managed by a hardware security module (HSM) for secure storage. To allow the usage of a commercially available HSMs it's recommended for the signing process to be able to support the PKCS#11 standard[3].

Note that in case security keys do get lost/stolen/etc it is possible for some devices to revoke or update the valid set of keys. However this can be quite limited e.g. on i.MX6 device one can *one-time* program up to four acceptable keys and each of those can be flagged as revoked, but it's impossible to add more or replace any keys.

[3] https://en.wikipedia.org/wiki/PKCS_11

Apertis secure boot integration

Integrating secure boot into Apertis really exists out of two parts. The first part is to ensure all boot stages have the ability to verify. The second part is to be able to sign all the boot stages as part of the Apertis image building process. While the actual implementation details of both will be system/hardware/SoC specific the impact of this is generic for all.

As Apertis images are composed out of pre-build binary packages the package delivering the implementation for the various boot stages should either provide a build which will always enforce signature verification *or* the implementation should detect if the device is configured for secure boot and only enforce in that situation. Enforcing on demand has the benefit that it makes it easier to test the same builds on non-secure devices (though care must be taken that secure boot status cannot be faked).

For the signing of the various stages this needs to be done at image build time such that the signing key can be chosen based on the target. For example whether it's a final production build or a development build or even a production build to test on development devices. This in turn means that the signing tools and implementation need to support signing outside the build process which is normally supported.

Apertis secure boot implementation steps

As the whole process is somewhat device specific implementation of a secure boot flow for Apertis should be done on a device per device basis. The best starting point is is most likely the NXP i.MX6 sabrelite reference board as the secure boot process (Highly Assured Boot in NXP terms) is both well-known and well supported by upstream components. Furthermore an initial PoC for the early boot stages was already done for the NXP Sabre Auto boards which are based on the same SoC.

SabreLite secure boot preparation

The good introduction into HAB (High Assurance Boot)[4] is prepared by Boundary Devices, also there are some documentation[5] and examples in U-Boot source tree.

The NXP Code Signing Tool[6] is needed to create keys, certificates and SRK hashes used during the signing process –please refer to section 3.1.3 of CST User'

[4]https://boundarydevices.com/high-assurance-boot-hab-dummies/

[5]https://github.com/u-boot/u-boot/blob/master/doc/imx/habv4/introduction_habv4.txt

[6]https://gitlab.apertis.org/pkg/imx-code-signing-tool

s Guide[7]. Apertis reference images use the public git repository[8] with all secrets available, so it could be used for signing binaries during development in case if board has been fused with Apertis SRK hash (**irreversible operation!!!**).

Caution: the SabreLite board can be fused with the SRK (Super Root Key) hash only once!

To fuse the Apertis SRK hash[9] we have to have the hexadecimal dump of the hash of the key. Command below will produce the output with commands for Apertis SRK hash fusing:

```
$ hexdump -e '/4 "0x"' -e '/4 "%X""\n"' SRK_1_2_3_4_fuse.bin | for i in `seq 0 7`; do read h; echo fuse prog -
y 3 $i $h; done
```

This command generates the list of commands to be executed in a U-Boot CLI. For Apertis SRK hash fusing they are:

```
fuse prog -y 3 0 0xFD415383
fuse prog -y 3 1 0x519690F5
fuse prog -y 3 2 0xE844EB48
fuse prog -y 3 3 0x179B1826
fuse prog -y 3 4 0xEC0F8D7C
fuse prog -y 3 5 0x2F209598
fuse prog -y 3 6 0x9A98BE3
fuse prog -y 3 7 0xAAD9B3D6
```

After execution of commands above only Apertis development keys[10] can be used for signing the U-Boot binary.

The i.MX6 ROM does signature verification of the bootloader during startup, and depending on the configured (fused) mode the behaviour is different. The i.MX6 device may work in 2 modes:

- "open"–the HAB ROM allows the use of unsigned bootloaders or bootloaders signed with any key, without checking its validity. In case of errors, it will only generate HAB secure events on boot without halting the process. This mode is useful for development.
- "closed"–only signed with correct key U-Boot may be started, any incorrectly signed bootloader will not be started. This mode should be used only for final product.

It is highly recommended not to use "closed"mode for development boards!

[7]https://gitlab.apertis.org/pkg/imx-code-signing-tool/-/blob/apertis/v2021dev2/docs/CST_UG.pdf
[8]https://gitlab.apertis.org/infrastructure/apertis-imx-srk
[9]https://gitlab.apertis.org/infrastructure/apertis-imx-srk/-/blob/master/SRK_1_2_3_4_fuse.bin
[10]https://gitlab.apertis.org/infrastructure/apertis-imx-srk/

To check if your device is booted with correctly signed bootloader, and SRK key is fused, just type this in the U-Boot CLI:

```
=> hab_status

Secure boot enabled

HAB Configuration: 0xcc, HAB State: 0x99
No HAB Events Found!
```

The output shows if the device is in “closed”mode (secure boot enabled) and booted without any security errors.

In case of errors in “open”mode the same command will show the list of HAB events similar to:

```
--------- HAB Event 5 -----------------
event data:
    0xdb 0x00 0x14 0x41 0x33 0x21 0xc0 0x00
    0xbe 0x00 0x0c 0x00 0x03 0x17 0x00 0x00
    0x00 0x00 0x00 0x50

STS = HAB_FAILURE (0x33)
RSN = HAB_INV_CERTIFICATE (0x21)
CTX = HAB_CTX_COMMAND (0xC0)
ENG = HAB_ENG_ANY (0x00)
```

During Linux kernel verification it is possible to emulate the “closed”mode with `fuse override` command and proceed with the boot:

```
=> fuse override 0 6 0x2
=> run bootcmd
```

Note: the only issue with closed mode emulation –the device will accept kernel signed with any key, but HAB events will be generated and shown in that case.

To close a device you need to fuse the same values used for overriding.

Caution: the board can only use bootloaders signed with the Apertis development key after the step below! This is irreversible operation:

```
=> fuse prog 0 6 0x2
```

Secure boot in the U-Boot package for Sabrelite

The U-Boot bootloader must be configured with the option `CONFIG_SECURE_BOOT` to enable support of HAB (High Assurance Boot) support on i.MX6 platform.

Upstream U-Boot has no protection based on the HAB engine to prevent executing unsigned binaries. Verified boot with the usage of HAB ROM is enabled in U-Boot for Apertis only for FIT (Flattened uImage Tree)[11] format since it allows to embed Linux kernel, initramfs and DTB into a single image. Hence the support of FIT images must be enabled in U-Boot configuration by option `CONFIG_FIT`.

The patch series[12] enables verification of FIT image prior to execution of the Linux kernel. Patched U-Boot do verification of the whole FIT binary prior to extraction kernel and initramfs images, and this ensures that only verified initial system will be started.

All other format types like zImage, as well as other boot methods are prohibited on fully secured device when “closed”mode is enabled or emulated.

Sign U-Boot bootloader such that the ROM can verify

To sign the U-Boot for SabreLite we need `cst` tool installed in the system and the Apertis development keys repository[13] need to be checked out. Please use the csf/csf_uboot.txt[14] file as a template for your U-Boot binary.

U-Boot for SabreLite board doesn't use SPL, hence the whole `u-boot.imx` binary must be signed. With enabled `CONFIG_SECURE_BOOT` option the build log will contain following output (and file `u-boot.imx.log` as well):

```
Image Type:   Freescale IMX Boot Image
Image Ver:    2 (i.MX53/6/7 compatible)
Mode:         DCD
Data Size:    606208 Bytes = 592.00 KiB = 0.58 MiB
Load Address: 177ff420
Entry Point:  17800000
HAB Blocks:   0x177ff400 0x00000000 0x00091c00
DCD Blocks:   0x00910000 0x0000002c 0x00000310
```

we need values from the string started with “HAB Blocks:”. Those values must be used in “[Authenticate Data]”section of template[15]:

```
[Authenticate Data]
    Verification index = 2
    Blocks = 0x177ff400 0x00000000 0x00091C00 "u-boot.imx"
```

To sign the U-Boot with `cst` tool simply call:

[11]https://github.com/u-boot/u-boot/blob/master/doc/uImage.FIT/source_file_format.txt
[12]https://gitlab.apertis.org/pkg/u-boot/-/merge_requests/4
[13]https://gitlab.apertis.org/infrastructure/apertis-imx-srk
[14]https://gitlab.apertis.org/infrastructure/apertis-imx-srk/-/blob/master/csf/csf_uboot.txt
[15]https://gitlab.apertis.org/infrastructure/apertis-imx-srk/-/blob/master/csf/csf_uboot.txt

```
cst -i csf_uboot.txt -o csf_uboot.bin
```

File `csf_uboot.bin` will contain signatures which should be appended to original `u-boot.imx` binary:

```
cat u-boot.imx csf_uboot.bin > u-boot.imx.signed
```

Sign U-Boot bootloader for loading via USB serial downloader

In case if something goes wrong and the system does not boot anymore it is still possible to boot with the help of USB serial downloaders[16], such as `imx_usb_loader` or `uuu`.

However the U-Boot must be signed in a slightly different way since some changes are done by ROM in runtime while loading binary. Please refer to section "What about imx_usb_loader?"of High Assurance Boot (HAB) for dummies[17] document.

The template csf_uboot.txt[18] for signing U-Boot to be loaded over serial downloader protocol should contain additional block in "[Authenticate Data]"section:

```
[Authenticate Data]
    Verification index = 2
    Blocks = 0x177ff400 0x00000000 0x00091C00 "u-boot.imx", \
             0x00910000 0x0000002c 0x00000310 "u-boot.imx"
```

With the help of mod_4_mfgtool.sh[19] script we need to store and restore DCD address from original `u-boot.imx` in addition to signing:

```
sh mod_4_mfgtool.sh clear_dcd_addr u-boot.imx
cst -i csf_uboot.txt -o csf_uboot.bin
sh mod_4_mfgtool.sh set_dcd_addr u-boot.imx
cat u-boot.imx csf_uboot.bin > u-boot.imx.signed_usb
```

Sign kernel images for U-Boot to load

After the successful startup of U-Boot we need to load the Linux kernel, initramfs and DTB file into the memory. All these bits must be verified before transferring control to the kernel. With FIT (Flattened uImage Tree)[20] format we can use single signed image with kernel, initramfs and DTB embedded, and

[16] https://community.nxp.com/docs/DOC-95604
[17] https://boundarydevices.com/high-assurance-boot-hab-dummies/
[18] https://gitlab.apertis.org/infrastructure/apertis-imx-srk/-/blob/master/csf/csf_uboot.txt
[19] https://storage.googleapis.com/boundarydevices.com/mod_4_mfgtool.sh
[20] https://github.com/u-boot/u-boot/blob/master/doc/uImage.FIT/source_file_format.txt

this allows to avoid “mix and match”attacks with mixed versions of kernel, initramfs, DTB and configuration.

The signing procedure for kernel images is split into 2 parts:

- preparation of the kernel image in FIT format
- sign FIT image

FIT image creation

U-Boot documentation[21] contains a lot of details and examples how to create FIT images for different purposes.

To embed all bits into the single FIT image we need to prepare file in image tree source format, for Apertis we use simple template[22] containing configuration with 3 entries for kernel, initramfs and DTB respectively. So values `{{kernel}}`, `{{ramdisk}}` and `{{dtb}}` should be substituted with absolute or relative path to corresponding files.

Please pay attention to addresses in `load` fields, since the whole FIT image is loaded into the memory by address `0x12000000` (check the value of `kernel_addr_r` in U-Boot environment), it is important to avoid intersections with embedded binaries since they will be copied to configured memory regions after successful verification.

To create the FIT image you need to have `mkimage` command from the package `u-boot-tools` compiled with FIT support. With FIT source file prepared just run `mkimage` and generate the FIT binary:

```
$ mkimage -f vmlinuz.its vmlinuz.itb
FIT description: Apertis armhf kernel with dtb and initramfs
Created:         Fri Mar 13 02:23:33 2020
 Image 0 (kernel-0)
  Description:  Linux Kernel
  Created:      Fri Mar 13 02:23:33 2020
  Type:         Kernel Image
  Compression:  uncompressed
  Data Size:    4526592 Bytes = 4420.50 KiB = 4.32 MiB
  Architecture: ARM
  OS:           Linux
  Load Address: 0x10800000
  Entry Point:  0x10800000
  Hash algo:    sha1
  Hash value:   8a64994bdab06d01450560ea229c9f44f1f0af14
 Image 1 (ramdisk-0)
  Description:  ramdisk
```

[21]https://github.com/u-boot/u-boot/tree/master/doc/uImage.FIT
[22]https://gitlab.apertis.org/infrastructure/apertis-image-recipes/-/blob/apertis/v2023/sign/imx6/fit_image.template

```
  Created:      Fri Mar 13 02:23:33 2020
  Type:         RAMDisk Image
  Compression:  uncompressed
  Data Size:    20285185 Bytes = 19809.75 KiB = 19.35 MiB
  Architecture: ARM
  OS:           Linux
  Load Address: 0x15000000
  Entry Point:  unavailable
  Hash algo:    sha1
  Hash value:   c12652573d1b301b191cf3e2a318913afc1ae4b7
 Image 2 (fdt-0)
  Description:  Flattened Device Tree blob
  Created:      Fri Mar 13 02:23:33 2020
  Type:         Flat Device Tree
  Compression:  uncompressed
  Data Size:    42366 Bytes = 41.37 KiB = 0.04 MiB
  Architecture: ARM
  Hash algo:    sha1
  Hash value:   ace0dd1dea00568b1c4e6df3fb0420c912e3e091
 Default Configuration: 'conf-0'
 Configuration 0 (conf-0)
  Description:  Boot Apertis
  Kernel:       kernel-0
  Init Ramdisk: ramdisk-0
  FDT:          fdt-0
  Hash algo:    sha1
  Hash value:   unavailable
CSF Processed successfully and signed data available in vmlinuz.itb
```

Signing the FIT image

Now it is time to sign the produced image. The procedure is similar to signing U-Boot with additional step –we need to add the **IVT** (Image Vector Table) for the kernel image. We skip this step for U-Boot since it is prepared automatically during the build of the bootloader.

The IVT is needed for the HAB ROM and must be the part of the binary, it should be aligned to `0x1000` boundary. For instance, if the produced binary is:

```
$ stat -c "%s" vmlinuz.itb
25555173
```

we need to pad the file to nearest aligned value, which is `25559040`:

```
$ objcopy -I binary -O binary --pad-to=25559040 --gap-fill=0x00 vmlinuz.itb vmlinuz-
pad.itb
```

The next step is IVT generation for the FIT image and the easiest method is to use the `genIVT` script[23] provided by Boundary Devices with adaptation for padded FIT image:

- Jump Location –0x12000000 Here we expect the image will be loaded by U-Boot
- Self Pointer –0x13860000 (Jump Location + size of padded image) Pointer to the IVT table itself, which will place after padded image
- CSF Pointer –0x13860020 (Jump Location + size of padded image + size of IVT) Pointer to signature data, which we will add after IVT

So, the IVT generation is pretty simple:

```
$ perl genIVT
```

it will generate the binary named `ivt.bin` to be added to the image:

```
$ cat vmlinuz-pad.itb ivt.bin > vmlinuz-pad-ivt.itb
```

We need to prepare the config file for signing the padded FIT image with IVT. This step is absolutely the same as for U-Boot signing.

Configuration file for FIT image is created from template csf_uboot.txt[24], and values in `[Authenticate Data]` section must be the same as used for IVT calculation –Jump Location and the size of generated file:

```
[Authenticate Data]
    Verification index = 2
    # Authenticate Start Address, Offset, Length and file
    Blocks = 0x12000000 0x00000000 0x1860020 "vmlinuz-pad-ivt.itb"
```

At last we are able to sign the prepared FIT image:

```
$ cst -i vmlinuz-pad-ivt.csf -o vmlinuz-pad-ivt.bin
CSF Processed successfully and signed data available in vmlinuz-pad-ivt.bin
```

Signing bootloader and kernel from the image build pipeline

Starting with v2021dev1 Apertis uses single signed FIT kernel image for OSTree-based systems. The signed version of U-Boot is a part of U-Boot installer.

For signing binaries with the `cst` tool we need some files from the Apertis development keys[25] git repository. The minimal working setup should include only 6 files:

- `SRK_1_2_3_4_table.bin` –Super Root Keys table
- `key_pass.txt` –file with password

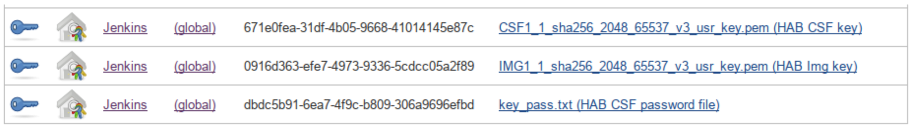
[23]https://storage.googleapis.com/boundarydevices.com/genIVT
[24]https://gitlab.apertis.org/infrastructure/apertis-imx-srk/-/blob/master/csf/csf_uboot.txt
[25]https://gitlab.apertis.org/infrastructure/apertis-imx-srk

- CSF certificate and key in PEM format
- IMG certificate and key in PEM format

In addition we need a template for the FIT source file and CSF template suitable for signing U-Boot and FIT kernel.

All files listed above are added into the git repository inside sign/imx6[26] subdirectory. Since all secrets for Apertis are public we are able to use them directly from the repo. However this is not acceptable for production.

Fortunately the most of CI tools have possibility to add files as secrets available only on several steps. Hence we add "private"keys and password file as "Secret file"global credentials to demonstrate the integration into the Jenkins pipeline:



		Jenkins	(global)	671e0fea-31df-4b05-9668-41014145e87c	CSF1_1_sha256_2048_65537_v3_usr_key.pem (HAB CSF key)
		Jenkins	(global)	0916d363-efe7-4973-9336-5cdcc05a2f89	IMG1_1_sha256_2048_65537_v3_usr_key.pem (HAB Img key)
		Jenkins	(global)	dbdc5b91-6ea7-4f9c-b809-306a9696efbd	key_pass.txt (HAB CSF password file)

For keys usage they should be available during the call of `cst` tool, so we have to add into the Jenkins pipeline copying of these secret files with the same names as used in CSF template[27] and remove them after the usage.

For instance the simple secrets copying for Jenkins:

```
withCredentials([ file(credentialsId: csf_csf_key, variable:  'CSF_CSFKEY'),
  file(credentialsId: csf_img_key, variable:  'CSF_IMGKEY'),
  file(credentialsId: csf_key_pass, variable: 'CSF_PASSWD')]) {
    // Setup keys for cst tool from Jenkins secrets
    // Have to keep keys and password file near certificates
    sh(script: """
      cd ${WORKSPACE}/sign/imx6
      cp -af $CSF_CSFKEY ./
      cp -af $CSF_IMGKEY ./
      cp -af $CSF_PASSWD ./""")
}
```

U-Boot signing

To sign the U-Boot the script scripts/sign-u-boot.sh[28] has been added. It automatically generates the CSF configuration from the template sign/imx6/fit_image_csf.template[29] and call the `cst` tool to sign the U-

[26]https://gitlab.apertis.org/infrastructure/apertis-image-recipes/-/tree/apertis/v2023/sign/imx6

[27]https://gitlab.apertis.org/infrastructure/apertis-image-recipes/-/blob/apertis/v2023/sign/imx6/fit_image_csf.template

[28]https://gitlab.apertis.org/infrastructure/apertis-image-recipes/-/blob/apertis/v2023/scripts/sign-u-boot.sh

[29]https://gitlab.apertis.org/infrastructure/apertis-image-recipes/-/blob/apertis/v2023/sign/imx6/fit_image_csf.template

Boot binary.

The script is called by the Debos recipe for the SabreLite U-Boot installer image[30]:

```
  - action: run
    description: Sign U-Boot
        script:  scripts/sign-u-boot.sh  "${ROOTDIR}/deb-binaries/usr/lib/u-
boot/{{ $target }}/u-boot.imx"
```

FIT image creation and signing

The FIT image is more complex. So for Apertis we use 2 scripts:

- the `scripts/generate_signed_fit_image.py` script[31] is used for generation FIT image, padding, IVT calculation and signing. This script can be used standalone to automate all steps described in the section "Sign kernel images for U-Boot to load"
- the `scripts/generate_fit_image.sh` script[32] is a wrapper for the former providing it the paths for kernel, initramfs and DTB to include them in the signed FIT image.

The integration with the build pipeline happens **after** the kernel is installed by the OSTree commit recipe[33] by adding the step below:

```
  - action: run
    description: Generate FIT image
    script: scripts/generate_fit_image.sh
```

NB: this action must be done prior to ostree commit action to add the signed FIT kernel into OSTree repository for OTA upgrades.

As next steps the following could be undertaken:

- Integration of PCKS#11 support in the signing process to support HSM devices
- Automated testing of secure boot if possible

[30]https://gitlab.apertis.org/infrastructure/apertis-image-recipes/-/blob/apertis/v2023/mx6qsabrelite-uboot-installer.yaml
[31]https://gitlab.apertis.org/infrastructure/apertis-image-recipes/-/blob/apertis/v2023/scripts/generate_signed_fit_image.py
[32]https://gitlab.apertis.org/infrastructure/apertis-image-recipes/-/blob/apertis/v2023/scripts/generate_fit_image.sh
[33]https://gitlab.apertis.org/infrastructure/apertis-image-recipes/-/blob/apertis/v2023/ostree-commit.yaml