Dependable Boot Specification

Release 0.1-alpha

Linaro Limited and Contributors

Nov 23, 2021

CONTENTS

CHAPTER

ONE

ABOUT THIS DOCUMENT

1.1 Introduction

Firmware updates in the Arm ecosystem have been historically handled by proprietary methods. Some level of firmware behavior and lifecycle standardization is required to open business opportunities in a broader ecosystem.

Part of the firmware lifecycle is the update process. Firmware and firmware update technologies are very platform dependent but the process can be standardized assuming that firmware will cover all the platform specific aspects.

The firmware update is driven by the main operating system through the standard UEFI update capsule technology, to ensure the update process is independent from the processor and OS implementation. The creation of update capsules and their signature(s) are outside the scope of the document.

This document addresses the robustness of the update process in order to provide:

- Brick protection (not limited to firmware upgrades)
- Rollback capabilities (if permitted)
- Rollback protection

While it is assumed that, in real life scenarios, there will be a lot of testing prior to deploying an update, unexpected behavior can still happen due to hardware revision differences, language configuration, operational conditions (power, temperature…) amongst other reasons. For the reasons above committing the updates and allowing the new firmware to be used must be verified and approved by the OS. Each vendor can have his application verifying the firmware operation before accepting it. This is going to be referred as 'transactional updates' for the rest of the document.

The intent of this document is to provide guidelines for firmware updates that will protect the device against bricking and rollback attacks. The update itself may occur from the non-secure world (Non-secure firmware) or the secure world (Secure firmware) depending on the implementation and the hardware requirements.

Comments or change requests can be sent to `boot-architecture@lists.linaro.org`.

1.2 Scope

Dependable Boot was written as a response to the lack of firmware upgrade standardization in the embedded system ecosystem. As embedded systems are becoming more sophisticated and connected, it is becoming increasingly important for embedded systems to standardize the way they upgrade their firmware. The document aims at protecting devices against unauthorized updates, defective updates and hardware failures.

1.3 Requirements and Key concepts

1.3.1 Prerequisites

Since this document targets [EBBR] and [PSBG] compliant systems, the update process of platforms without UEFI is outside the scope of this document.

- Every firmware binary **must** support dual A/B partitions for all of it's components.
- The first stage bootloader **must** be able to access our firmware NVCounter.
- UEFI capsule updates are the mechanism used for delivering firmware updates. Refer to[1] for details.
- We assume that our immutable stage loader can load the second stage boot loader (e.g TF-A) from a dual image. In case the masked ROM can not boot with an alternative location, similar functionality **must** be added to the secondary bootloader stage. In that case the secondary boot loader inherits all the requirements of the first stage boot loader and the secondary bootloader stage is disallowed from OTA updates. In person update of secondary bootloader stage may be possible and may require extra tools or procedures but that is out of the scope of this document.
- Update process can target a single firmware component or multiple components. Updating multiple components and multiplexing boot combinations can be very challenging. In this document we treat the firmware as a single entity regardless of the components it comprises. Failing to update one of the components will result in an overall failure, forcing the firmware to remain on the current bank.
- Updating the firmware and the OS at the same time is prohibited.
- A hardware watchdog **must** always be active when critical components are executed. It's advisable the watchdog is activated on the earliest possible boot stages. The watchdog **must** reset the board if a timeout occurs.

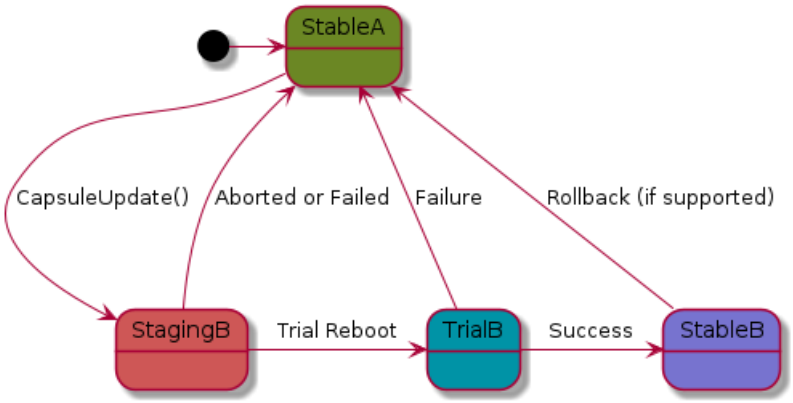
1.3.2 Transactional updates

Since firmware updates are independent from the OS upgrades, an important aspect of the update policy is the device functionality. If the OS ends up being unusable after a firmware update (while the platform is in the Trial state [FWU]), the device must be able to detect this and force the platform to the previously working firmware. Two scenarios can be considered:

1. The FW acceptance tests, executed by the OS, fail: The OS must request a FW downgrade by using the methodology described in *OS requested FW revert*.
2. The OS fails to boot, e.g. due to a watchdog timer fire: The Normal World FW must keep a counter of the attempted OS boots in the Trial State. If the platform is in the Trial State after a platform specific number of OS boot attempts, the Normal World FW must take the steps to revert the FW to the previous active FW bank.

If the FW acceptance tests, executed by the OS, pass, then the newly installed FW can be made permanent. To mark the FW as permanent the OS will install an acceptance capsule as described in *OS directed FW image acceptance*. The Non-secure firmware will, on next reboot, mark all FW images in the active bank as accepted and update the rollback counters to the new values (if applicable) [FWU]. Once all the FW images in the active bank are accepted, the device transitions from the Trial state to the Regular state [FWU].

[1] [UEFI] 2.8B § 23 - Firmware Update and Reporting



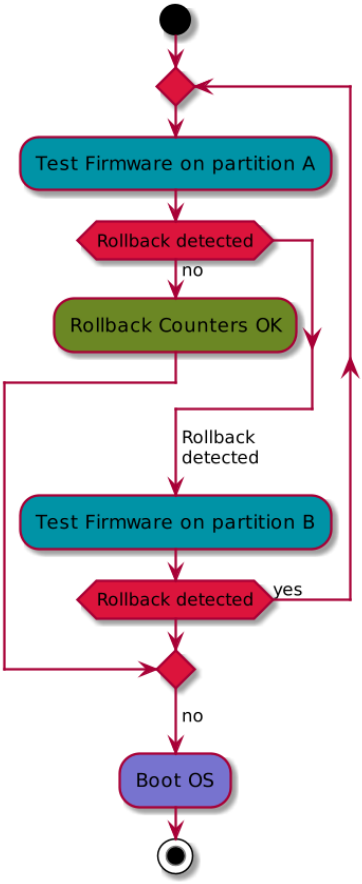
1.3.3 Rollback protection

Nodes need to provide protection against rollback based attacks. If at any point the device firmware was updated to patch security vulnerabilities, allowing rollback to any previous insecure versions is a security risk. It's possible for a non-persistent attack to download, flash an older vulnerable version of the firmware and install a permanent exploit on the device. Downgrading to a previous version should be prohibited but in case an older version is detected the device can check the backup partitions. If a valid version is found on the secondary partition the device can reboot using that. If both partitions are invalid the device will go into a reboot loop.

1.3.4 Brick protection

The brick protection mechanism not only addresses the bricking during the firmware update, but can optionally protect the device against hardware failures. The rollback counter might or might not be always bumped during an update. If the secondary partition contains a valid firmware and the primary partition is unable to boot the device (e.g flash corruption), the device is allowed to fallback on the secondary partition.

If the update is going to bump the rollback counters it's strongly advised to update both of the partitions. In that case the upgrade process will run once to update the secondary partition. Once that's finished and accepted, the firmware update agent should update the former primary partition as well. This process must not necessarily go through the entire update procedure. Simply writing and verifying the firmware is enough. Similarly if an update of the secondary partition fails (e.g write failed), then the firmware update agent must restore the partition to a known working state (e.g overwrite it with the current working firmware)


Test Firmware on partition A
Rollback detected
no
Rollback Counters OK
Rollback
detected
Test Firmware on partition B
Rollback detected
yes
no
Boot OS

CHAPTER

TWO

UEFI

This chapter discusses specific UEFI implementation details.

2.1 UEFI Version

This document uses version 2.9 of the UEFI specification [UEFI].

2.2 UEFI Compliance

All [UEFI] features required by [EBBR] are assumed.

2.3 UEFI extra dependencies

The capsules installed using the procedure defined in this document must be formatted according to the FMP defined format [UEFI] § 23.1 - Firmware Management Protocol. The FMP instance must provide the GetImageInfo and SetImage functions in order to be used as the UpdateCapsule backend.

2.3.1 ESRT

[UEFI] defines the System Resource Table (ESRT) § 23.4 - EFI System Resource Table.

Each entry in the ESRT describes a device or system firmware resource that can be targeted by a firmware capsule update. Each entry in the ESRT will also be used to report status of the last attempted update.

The UEFI specification defines a mapping between the ESRT fields and the EFI_FIRMWARE_IMAGE_DESCRIPTOR provided by FMP.GetImageInfo().

The resource entry field FwClass must be set to "ESRT_FW_TYPE_SYSTEMFIRMWARE" The fields LowestSupportedVersion, FwVersion are provided by - the Secure World FWU when Secure world is responsible for the update - Non-secure firmware if Non-Secure world is responsible for the update

LastAttemptStatus is expected to be information kept by the UEFI implementation. This can partially rely on information provided by early platform bootstages [FWU].

The acceptance status of each FW image is provisionally displayed in the LastAttemptedStatus field[1] in the ESRT image entry. A value of 0x3fff implies that the image has not been accepted. The OS must explicitly accept the image by installing an acceptance capsule as described in *OS directed FW image acceptance*.

A UEFI implementation can opt to implicitly accept any FW image. It is strongly recommended that the FW image acceptance is treated as an OS responsibility.

[1] Presenting the image acceptance status in the LastAttemptedStatus field is a provisional arrangement. A more permanet solution is under discussion.

2.3.2 OS requested FW revert

The UEFI implementation informs the OS of the current FW images via the ESRT.

The OS can request a FW downgrade. The FW downgrade request is a hint to UEFI that the FW bank should be moved to a previously working bank.

If the OS wants to revert the FW images to a previously working bank, it can do so by installing the following signed capsule:

- CapsuleGuid = acd58b4b-c0e8-475f-99b5-6b3f7e07aaf0
- HeaderSize = sizeof(EFI_CAPSULE_HEADER)
- Flags = 0
- CapsuleImageSize = sizeof(EFI_CAPSULE_HEADER)

Note: the image acceptance capsule must be authenticated. Details TBD.

When UEFI receives the capsule above, UEFI will change the FWU metadata active_index to a previously working bank index by either:

1. Calling the FWU primitive fwu_set_active [FWU] if the flash store is owned the Secure World.
2. Set the active_index field in the FWU Metadata [FWU] if the flash store is owned by the Normal World.
3. Restore EFI variables to their previous state, if changed during the firmware upgrade.

2.3.3 OS directed FW image acceptance

OS declaration of future capsule acceptance responsibility

The OS can inform the UEFI implementation of all the FW images that the OS intends to explicitly accept.

The OS can set the EFI_CAPSULE_HEADER.Flags[15] to 1 indicating that it will explicitly accept all the FW images contained in this capsule. The OS must be able to accept all images that it registers to accept. Any other images should be implicitly accepted by the UEFI implementation.

FW image acceptance

The OS must accept each image, that has an acceptance pending, by using a capsule composed of an EFI_CAPSULE_HEADER concatenated with the image type UUID:

- CapsuleGuid 0c996046-bcc0-4d04-85ec-e1fcedf1c6f8
- HeaderSize = sizeof(EFI_CAPSULE_HEADER)
- Flags = 0
- CapsuleImageSize = sizeof(EFI_CAPSULE_HEADER) + sizeof(UUID)

Note: the image acceptance capsule must be authenticated. Details TBD.

2.3.4 Update permission verification

The FW management guidelines in [NIST_800_193] specify that the system should check:

1. FW image authenticity.
2. FW update procedure authorization.

The FW image authenticity should be implemented by authenticating the different FW images. The FW update authorization should be implemented by verifying that the capsule or its components were assembled by the platform owner.

FW update authorization

The FW update authorization [NIST_800_193] can be checked by the OS, using OS specific methods, before calling UpdateCapsule. Alternatively, the FW update authorization can rely on the FW image authenticity check. If all FW images in the capsule are authentic then the user is deemed authorized to progress with the FW update procedure.

FW image authentication

Each FW image should be signed by the FW vendor. The mechanism for FW image vendor public key to be provisioned is outside the scope of this document.

The FW vendor signature should be placed before the FW image as is described in the UEFI FMP definition (§ 23.1 [UEFI]).

The FW images should be authenticated before being written to the FW store or before being allowed to execute on the platform.

2.3.5 Maximum Trial platform boots

The UEFI implementation must keep a count of the consecutive platform boots in the Trial state [FWU]. If the number of consecutive platform boot in the Trial state exceeds a platform defined value of *max_trial_boots* then the UEFI implementation must revert the FW to the previous working bank [FWU].

Note: Similar functionality must be implemented in the first stage boot loader. This is platform specific, but the first stage bootloader must be able to count the number of reboots. If the number exceeds *max_trial_boots* then we must revert to a previous working version of the firmware.

CHAPTER

THREE

SECURE FIRMWARE ON NON-SECURE FIRMWARE UPDATES

3.1 Firmware Flash owned by Secure World

If the flash device is owned by the Secure World, the FMP managing the FW images must communicate the FW images to the Firmware Update Implementation [FWU]. The Firmware Update Implementation in Secure World [FWU] writes the FW images to flash. In this model the OS can install the capsule by invoking the UEFI UpdateCapsule runtime service. The capsule can be installed, by the UEFI Implementation, without requiring a system reboot.

The FWU metadata [FWU] is managed by the FWU Implementation in the Secure world. The FWU metadata is described in Section 4.1 of [FWU].

When the flash is owned by the Secure World, the FMP communicates with the FWU Implementation in the secure world using the FF-A Firmware Update ABI [FWU]

The FWU Implementation provides Non-secure firmware with a list of all FW images handled by it. The information is provided via the image directory[1].

3.1.1 FMP interface for Secure World updates

The FMP is responsible for committing the FW images to flash and to provide the information used to construct the ESRT table.

Each image entry in the FWU image directory exposes a set of fields which map directly to the EFI_FIRMWARE_IMAGE_DESCRIPTOR as defined in the following table:

Table 3.1: EFI_FIRMWARE_IMAGE_DESCRIPTOR Implementation Requirements

FWU image directory entry field	EFI_FIRMWARE_IMAGE_DESCRIPTOR
img_type_uuid	ImageTypeId
lowest_accepted_version	LowestSupportedImageVersion
last_attempted_version	LastAttemptedVersion
version	Version
image_max_size	Size

[1] image directory is a stream of data structured as an array of image entries [FWU].

3.2 Firmware Flash owned by non-Secure World

If the flash device is owned by the Normal World, the FW images must be written to it directly by the UEFI implementation.

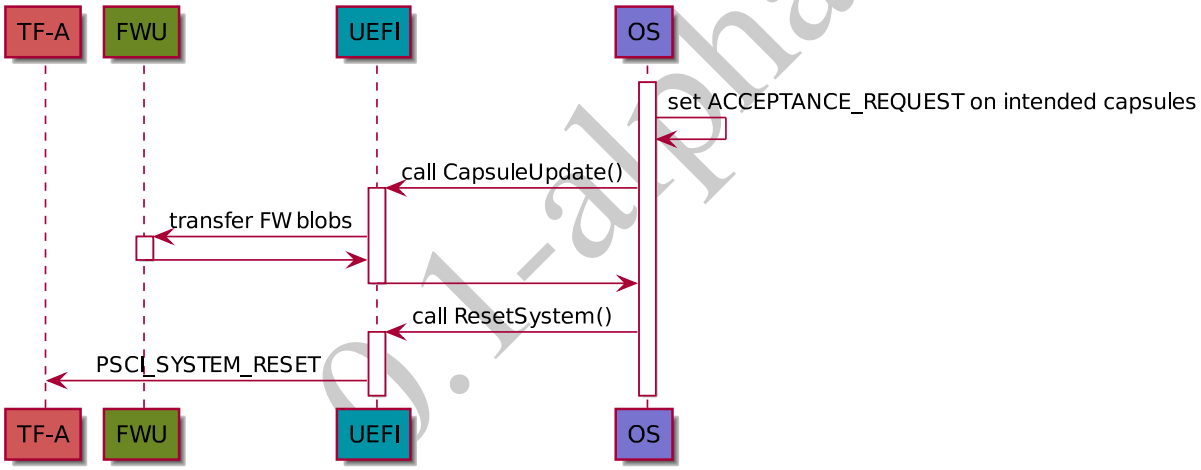
Two models exist in this platform model:

- The OS places the capsule on the EFI system partition in the /efi/updatecapsule directory, as defined in § 8.5.5 - Delivery of Capsules via file on Mass Storage device [UEFI]. After this the OS requests a platform reset. The OS may optionally install an update application which installs the capsule at the next reboot.
- The OS calls the UEFI UpdateCapsule runtime service. The Capsule must have the CAPSULE_FLAGS_PERSIST_ACROSS_RESET bit set in the EFI_CAPSULE_HEADER flags field. The implementation must correctly flush all caches prior to performing the warm reset.

The FWU metadata [FWU] is managed by the UEFI implementation. The FWU metadata is described in Section 4.1 of [FWU].

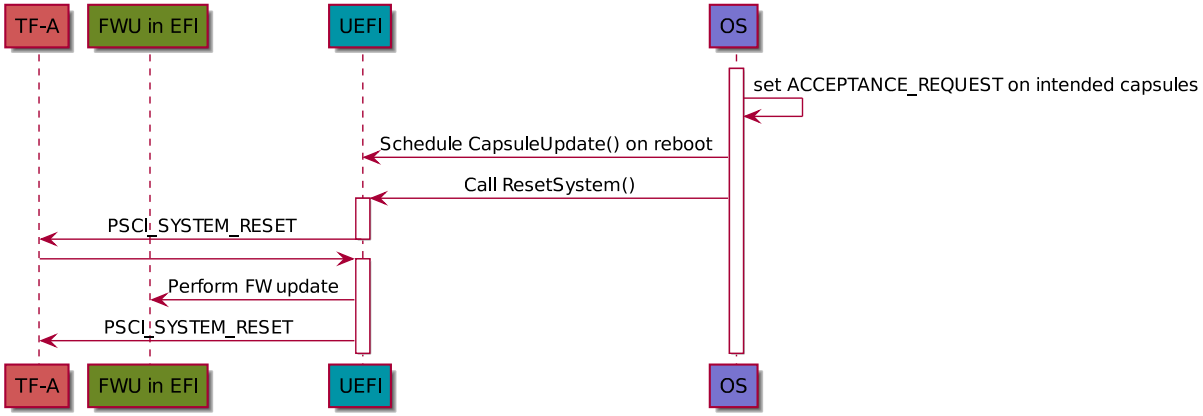
3.3 Example flows

3.3.1 Capsule install Secure World



1. OS receives a capsule with the new firmware
2. OS sets the acceptance request bit the Capsule header for all the images it wants to accept
3. OS passes the capsules to the UpdateCapsule runtime service
4. UEFI implementation traverses all the images in the capsule passing them to their corresponding FMPs
5. The FMP transfers the images to the FWU Implementation in the Secure World [FWU]
6. OS requests a system reboot

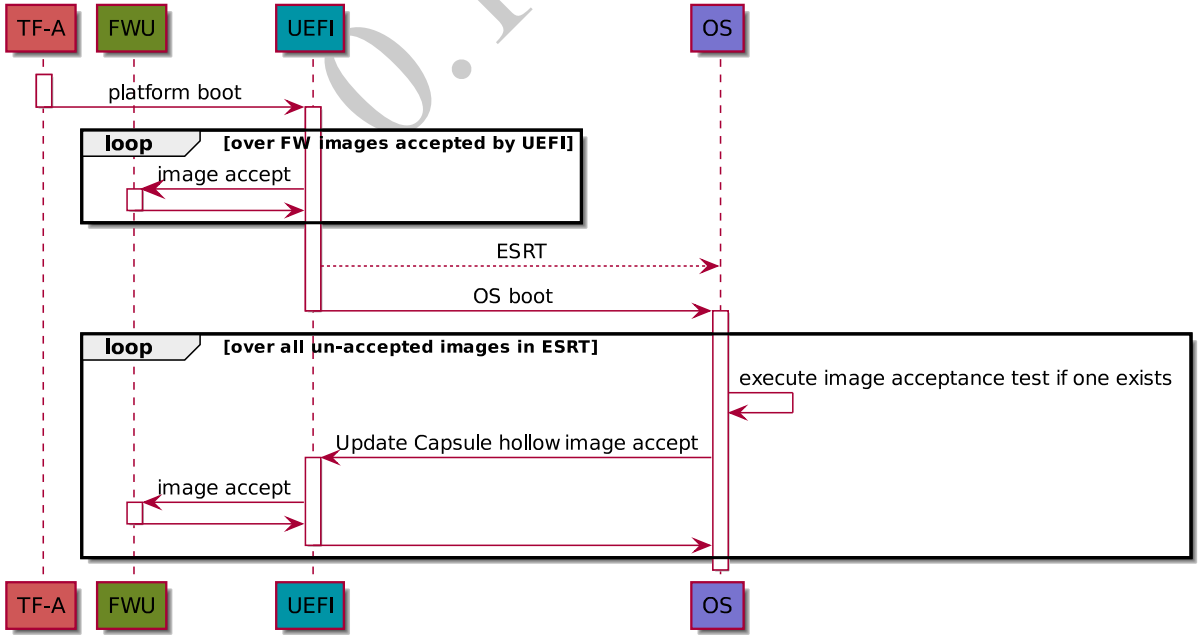
3.3.2 Capsule install non-Secure World



1. OS receives a capsule with the new firmware
2. OS sets the acceptance request bit the Capsule header for all the images it wants to accept
3. OS schedules a CapsuleUpdate on disk and reboots
4. UEFI implementation traverses all the images in the capsule passing them to their corresponding FMPs
5. The UEFI firmware performs the update
6. UEFI firmware requests a system reboot

3.3.3 Post-capsule install reboot success

Note: When the Normal world controls flash, FWU and UEFI are within the same execution context. In this case, the activations and returns between FWU and UEFI are internal to the UEFI implementation.

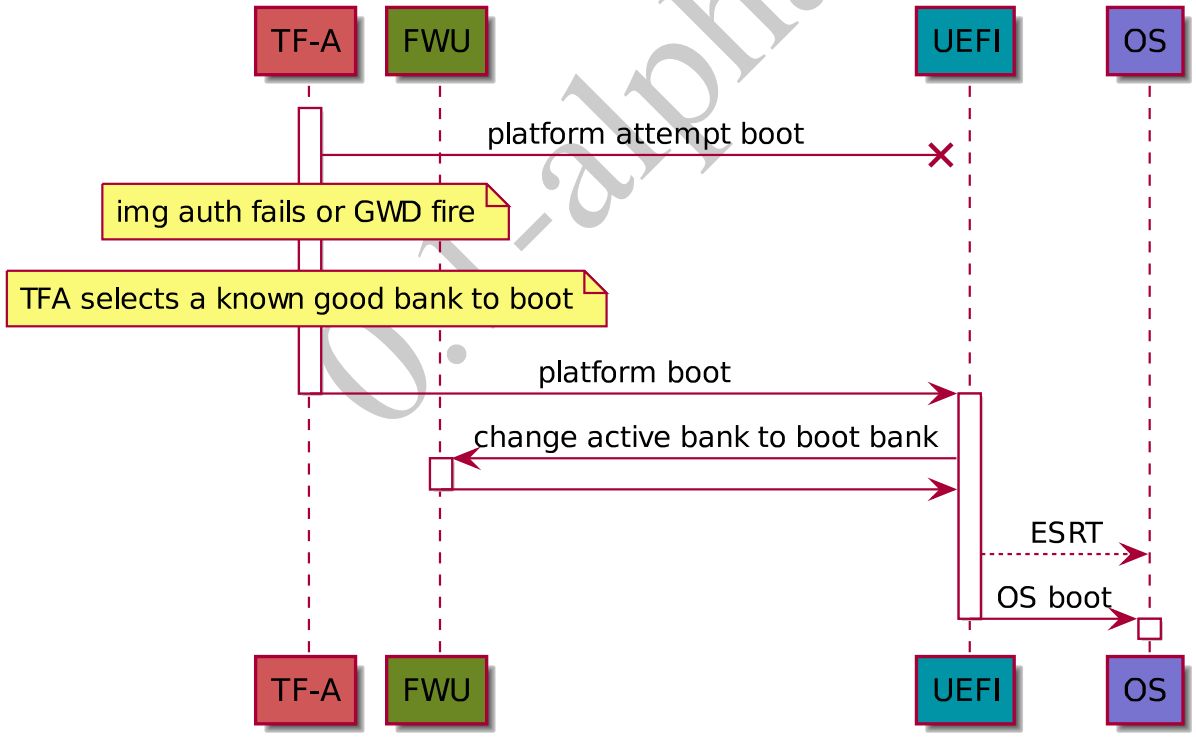


1. Platform boots with the new FW
2. From the TFA boot report [FWU], UEFI verifies that platform booted from the intended bank
3. UEFI accepts a sub-set of the FW images [FWU] (the sub-set is platform specific)

4. OS loader obtains the ESRT from UEFI
5. OS boots
6. OS inspects the information in the ESRT
7. OS performs an image acceptance test for any un-accepted image
8. If all image tests pass correctly the OS exits the FW update procedure
9. OS install the image acceptance capsule when all acceptance tests pass
10. Firmware processes the image acceptance capsule and updates the boot bank
11. **Rollback counter updates**
    - If the Non-secure firmware can update the rollback counter(s) directly, it should do it on the fly
    - Otherwise, on the next reboot Secure firmware must detect the new version (rollback counter < fw rollback counter)and update the rollback counter(s) accordingly.

3.3.4 Post-capsule install reboot fails before UEFI

Note: When the Normal world controls flash, FWU and UEFI are within the same execution context. In this case, the activations and returns between FWU and UEFI are internal to the UEFI implementation.

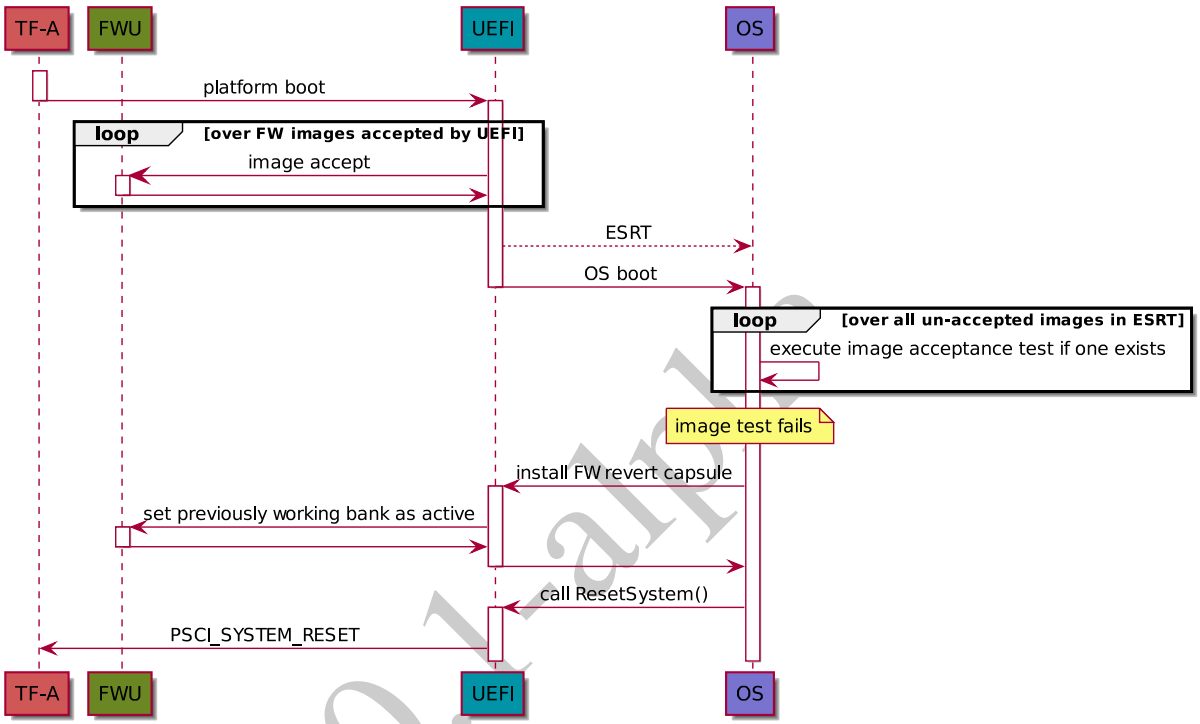


1. Platform boots with the new FW
2. The images fail to authenticate or the generic watchdog fires
3. Platform resets
4. Early platform bootloader detects FW maltfunction and selects another bank to boot from
5. UEFI receives the report from TFA of the failed boot attempt
6. UEFI effectivates the permanent bank change
7. UEFI generates the ESRT reflecting the bank that booted the system

8. OS loader obtains the ESRT from UEFI
9. OS boots
10. OS inspects the information in the ESRT

3.3.5 Post-capsule install image fails OS test

Note: When the Normal world controls flash, FWU and UEFI are within the same execution context. In this case, the activations and returns between FWU and UEFI are internal to the UEFI implementation.



1. Platform boots with the new FW
2. From the TFA boot report [FWU], UEFI verifies that platform booted from the intended bank
3. UEFI accepts all images [FWU]
4. OS loader obtains the ESRT from UEFI
5. OS boots
6. OS inspects the information in the ESRT
7. OS performs an image acceptance test for any un-accepted image
8. If any image tests fails, the OS install a “FW downgrade request” capsule, instructing UEFI to select the previously working FW bank, or imediately reboots.
9. OS requests a system reset

CHAPTER

FOUR

CONVENTIONS

4.1 Conventions Used in this Document

The key words “MUST”, “MUST NOT”, “REQUIRED”, “SHALL”, “SHALL NOT”, “SHOULD”, “SHOULD NOT”, “RECOMMENDED”, “MAY”, and “OPTIONAL” in this document are to be interpreted as described in **RFC 2119**.

4.2 Terms and abbreviations

This document uses the following terms and abbreviations.

UEFI Unified Extensible Firmware Interface.

Anti-brickable A system is said to be a brick if it cannot boot for a firmware behavior issue and no firmware update is possible. Anti-Brickable protection is a set of measures to protect against this risk for any firmware component.

Boot firmware Firmware that brings up operating system

EBBR Embedded Base Boot Requirements

ROM Read only memory

Immutable bootloader stage (Immutable stage) First bootloader stage, generally stored in a ROM – equatable to BL1 in TF-A or u-boot-spl.

Secondary bootloader stage (Secondary stage) Optional bootloader stage that executes following the immutable stage – equatable to BL2 in TF-A.

Monitor Firmware EL3 Runtime Firmware – refered to as BL31 in TF-A.

Secure firmware Firmware executing at S-EL2 and/or S-EL1 – BL32 in the TF-A context. The Secure payload can be composed of different binaries.

Non-secure firmware Firmware executing at EL2 or EL1 – BL33 in the TF-A context.

SCP Firmware System Control Processor firmware

PSGB Platform Security Boot Guide

Trusted Substrate Set of firmwares that control security and trust aspects of a platform. For instance device identity management firmware.

FFA Arm Firmware Framework for Armv-8A

BIBLIOGRAPHY

[UEFI] Unified Extensable Firmware Interface Specification v2.9, February 2020, UEFI Forum

[FFA] Arm Firmware Framework for Armv8-A, September 2018, Arm Limited

[FWU] Platform Security Firmware Update for the A-profile Arm Architecture 1.0, May 2021, Arm Limited

[EBBR] Embedded Base Boot Requirements v2.0.0-pre1, January 2021, Arm Limited

[PSBG] Platform Security Boot Guide, July 2020, Arm Limited

[NIST_800_193] Platform Firmware Resiliency Guidelines, May 2018, NIST

INDEX