Firmware Security

Roland Nagy
CrySyS Lab, HIT, BME
rnagy@crysys.hu

Agenda

- Intro
- Secure Boot
  - Goals & requirements
  - Crypto recap
  - Authentication schemes & hardware support
  - Example 1: boot flow of a microcontroller
  - Example 2: boot flow of UEFI
- Secure Firmware Update
  - Goals & requirements
  - A/B partitioning (Seamless Update) & rollbacks
  - Example 1: secure update of a microcontroller
  - Example 2: secure update of EUFI

Introduction

- What is the firmware?
  - “software for hardware”
  - “software that provides low level control for hardware”
  - “software that runs before the OS”

- For desktops/servers -> BIOS/UEFI
  - Starts a bootloader/OS

- For embedded devices, we often don’t have an OS
  - We call firmware everything that is code

- For network devices, again, we often call everything firmware
  - Despite having an OS, which in this case is part of the firmware image

Introduction

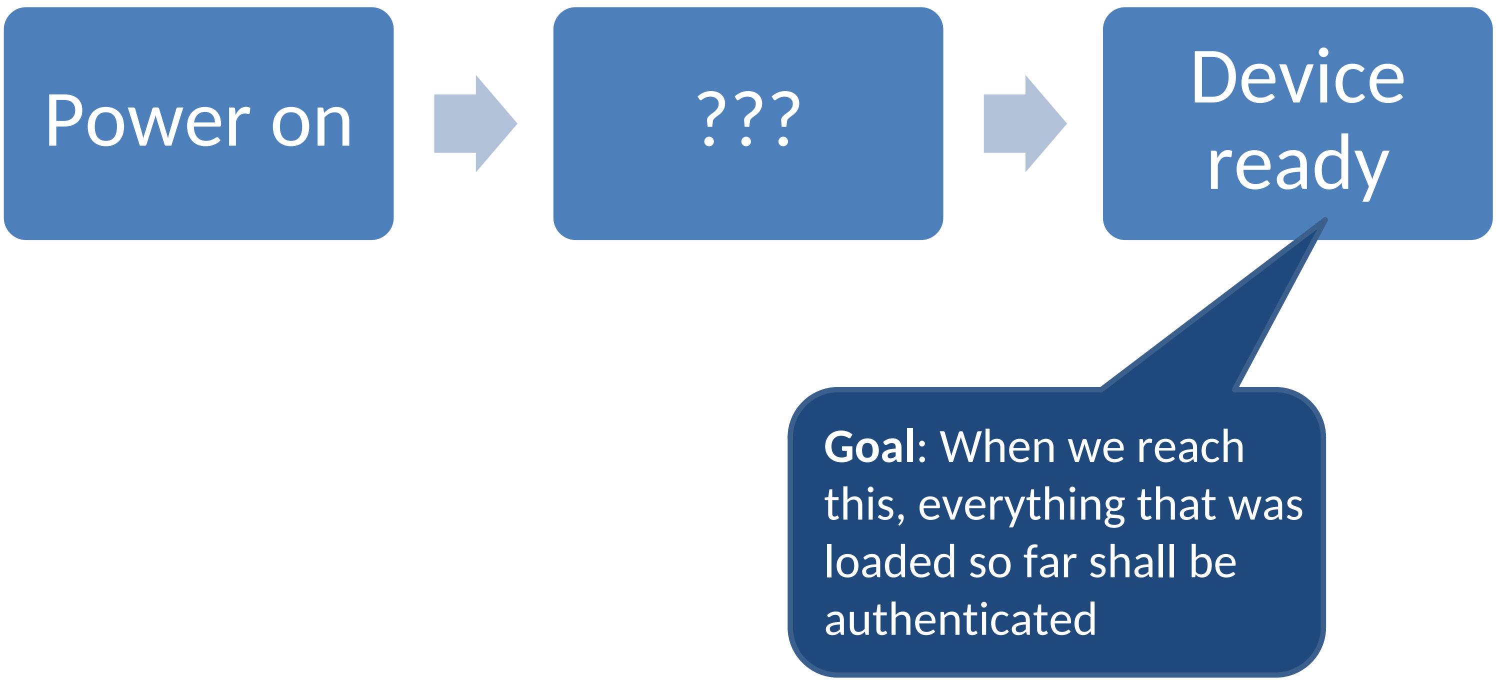
- So what is the firmware?
  - Code that runs when the device is starting
  - Perform hardware initialization
  - Starts other software components

- Why firmware security is important?
  - OS security features not available yet
  - If compromised, other components started by the firmware cannot be considered safe and secure either
  - After all, it's just code, it can have vulnerabilities
  - Modern firmwares can be quite large
  - Malware target it
    - Mebromi for BIOS
    - BlackLotus for UEFI
    - Boot sector viruses since 1986 (Brain, Lehigh, SCA, Ping-Pong virus, etc.)

Secure Boot

Secure Boot

- Very, very high level boot flow:



Secure Boot - Authentication

- To authenticate the loaded software component, we can use authentication schemes:
  - Hash-based authentication
    - Using hash functions
  - MAC-based authentication
    - Using Message Authentication Codes
  - Signature-based authentication
    - Using digital signatures

Cryptographic hash functions

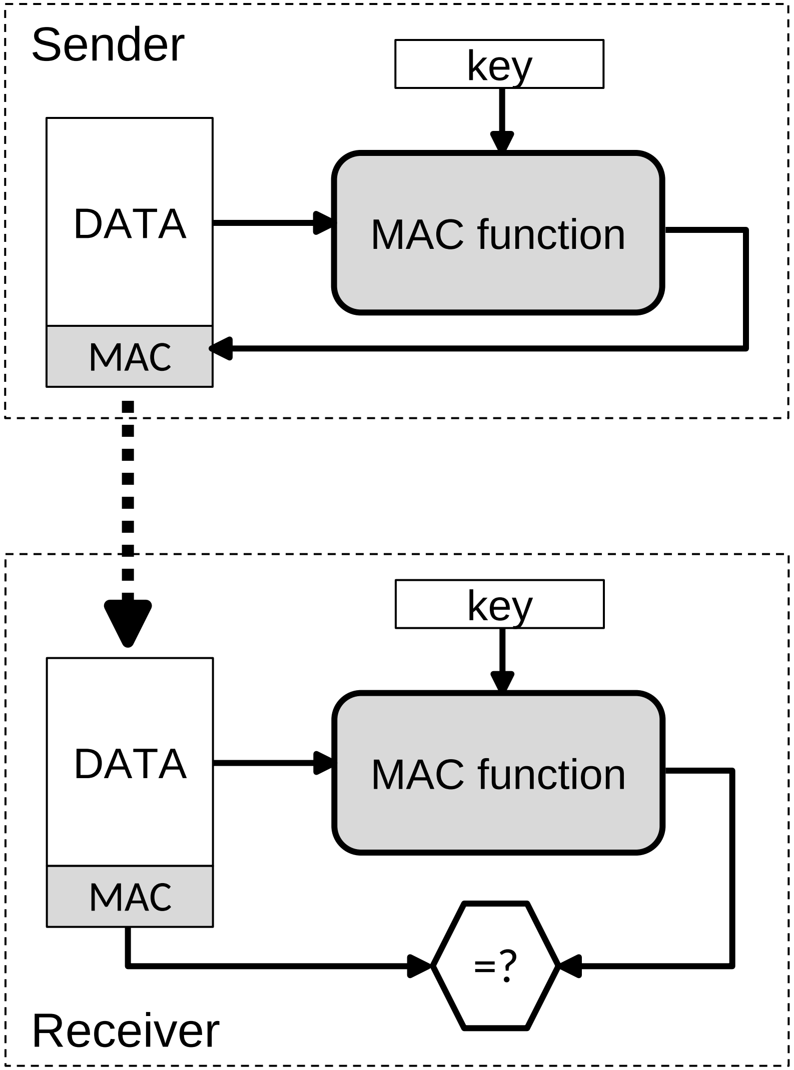
- a hash function is a function that maps arbitrary long messages into a fixed length output (n bits)
- notation and terminology:
  - x – (input) message
  - y = H(x) – hash value, message digest, fingerprint
- typical applications:
  - the hash value of a message can serve as a compact representative image of the message (similar to fingerprints)
  - increase the efficiency of digital signatures by signing the hash instead of the message (expensive operation is performed on small data)
  - password hashing
  - **checking if data was modified**
- examples:
  - (~~MD5, SHA-1~~) SHA-2 and SHA-3

Properties of crypto hash functions

- ease of computation
  - given an input x, the hash value H(x) of x is easy to compute
- **weak collision resistance** (2^{nd} preimage resistance)
  - given an input x, it is computationally infeasible to find a second input x' such that H(x') = H(x)
- **strong collision resistance** (collision resistance)
  - it is computationally infeasible to find any two distinct inputs x and x' such that H(x) = H(x')
- **one-way property** (preimage resistance)
  - given a hash value y (for which no preimage is known), it is computationally infeasible to find any input x such that H(x) = y
- collision resistant hash functions can typically be modeled as a random function (similar to block ciphers)

Message Authentication Codes (MAC)

- a MAC function is a function that maps an arbitrary long message and a key (k bits) into a fixed length output (n bits)
  - can be viewed as a hash function with an additional input (the key)
- services:
  - **message authentication and integrity protection:** after successful verification of the MAC value, the receiver is assured that the message has been generated by the sender and it has not been altered in transit
- examples:
  - HMAC, CBC-MAC, CMAC

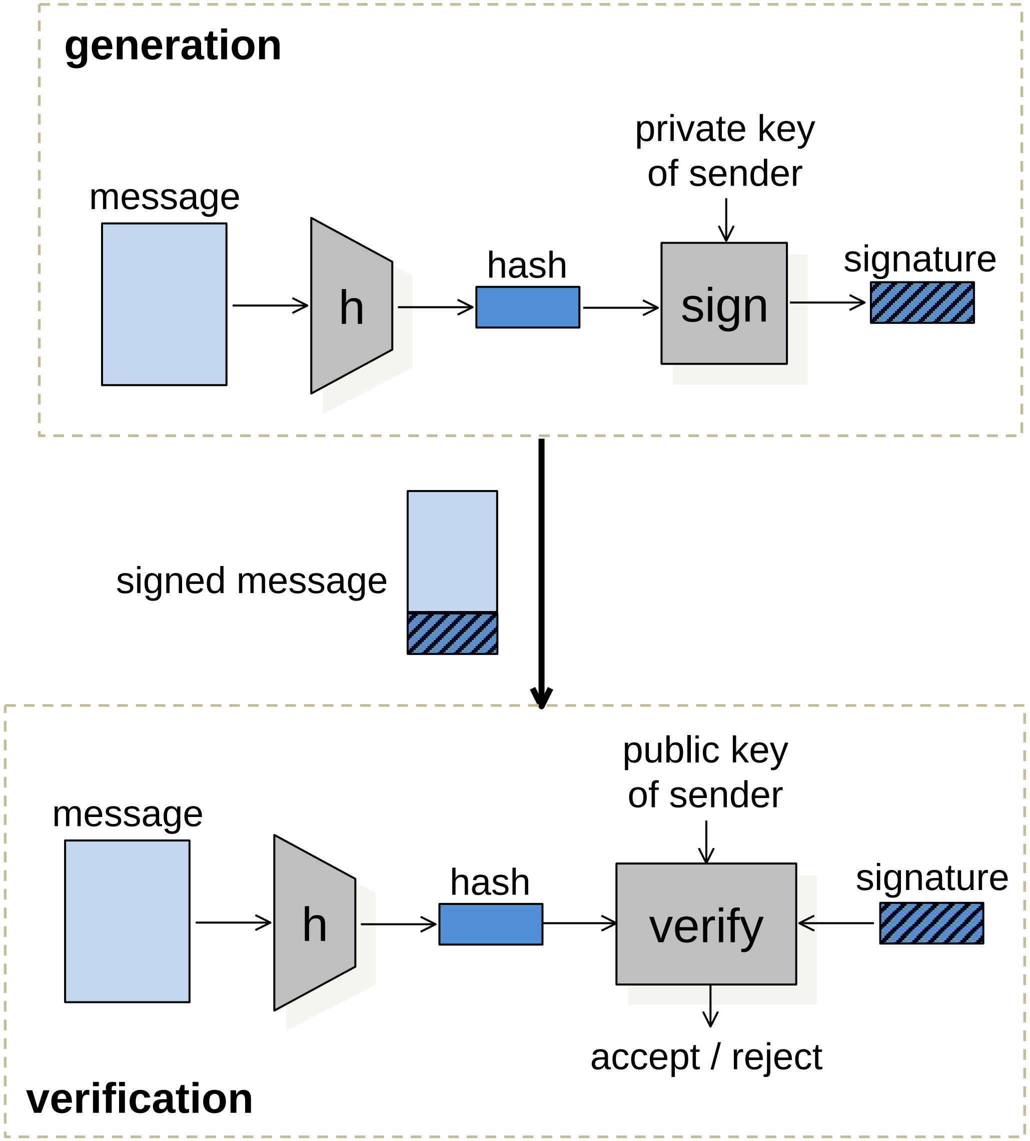


Digital signature schemes

- similar to MACs but they are
  - unforgeable by the receiver
  - verifiable by a third party
- services:
  - **message authentication and integrity protection:** after successful verification of the signature, the receiver is assured that the message has been generated by the sender and it has not been altered
  - **non-repudiation of origin:** the receiver can prove this to a third party (hence the sender cannot repudiate)
- examples: RSA, DSA, ECDSA

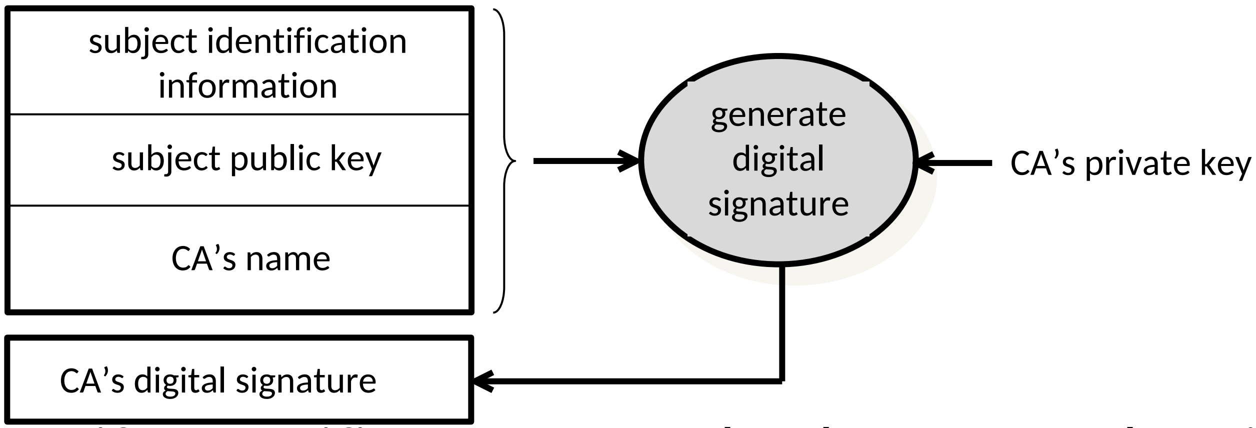
Hash-and-sign paradigm

- public/private key operations are slow
- increase efficiency by signing the hash of the message instead of the message
- it is essential that the hash function is collision resistant



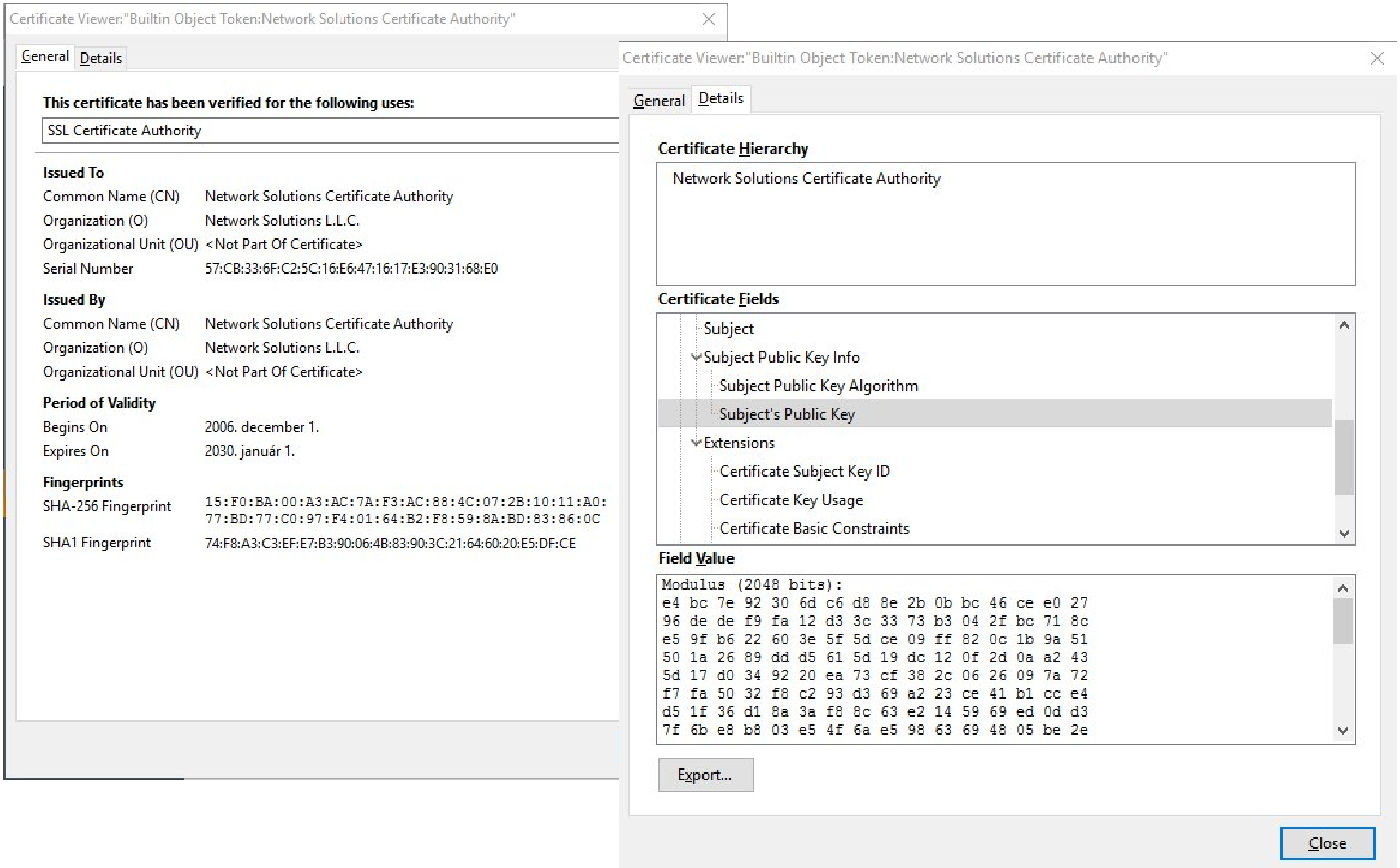
Basic idea of certificates

- name and public key is linked together by the digital signature of a **trusted entity** called **Certification Authority (CA)**



- in order to verify a certificate you need to have an authentic copy of the public key of the CA
- advantage: only the CA's public key need to be distributed via out-of-band channels (scales better)

Certificates illustrated



Certification Authority (CA)

- collection of hardware, software, and staff (people)
- main functions:
  - issues certificates for users or other CAs
  - maintains certificate revocation information
  - publishes currently valid certificates and certificate revocation lists (CRL)
  - maintains archives
- must comply with strict security requirements related to the protection and usage of its private keys (basis of trust)
  - uses tamper resistant Hardware Security Modules that enforce security policies (access and usage control)
  - defines and publishes its certificate issuing policies
  - complies with laws and regulations
  - is subject to regular control (by national supervising authority)

Secure Boot - Authentication

- To authenticate the loaded software component, we can use authentication schemes:
  - Hash-based authentication
    - Using hash functions
    - **Reference hash must be stored write protected**
  - MAC-based authentication
    - Using Message Authentication Codes
    - **Symmetric key must be stored write protected and confidential**
  - Signature-based authentication
    - Using digital signatures
    - **Signing public key, CA public key or their hash must be stored write protected**

Secure Boot – Hardware support

- To be able to use the authentication schemes, some hardware support is needed
  - One-time programmable memory (OTP)
    - Once written, after that, it's read-only
  - Secure Hardware Extension (SHE)
    - A cryptographic coprocessor
    - Can perform block cipher operations
    - Can store a symmetric key confidentially
  - Hardware Security Module
    - Can be a crypto coprocessor or a separate device
    - Can perform all sorts cryptographic operations
      - Hashing, symmetric/asymmetric encryption/decryption, MAC computation, etc.
    - Can store any type of keys & hashes
    - Usually has some sort of tamper protection

Secure Boot – Compatibility

- Different auth. schemes have different requirements, thus they are compatible with different hardware components

	Hash	MAC	Signature
OTP	Y*	N	Y**
SHE	N	Y	N
HSM	Y	Y	Y

*: Firmware image is not updatable
**: Image signing key or CA root certificate is not updatable

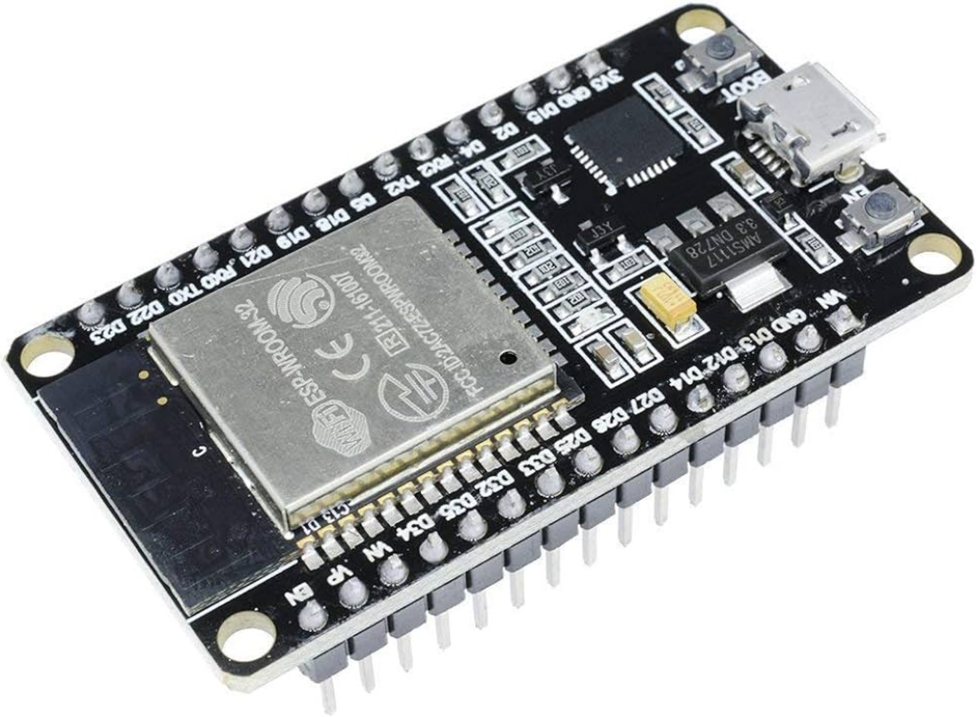
Secure Boot - RoT/CoT

- Root of Trust
  - A component we trust
  - Usually we are not checking its integrity
  - Trust is often provided by storing it in ROM

- Chain of Trust
  - The first component we trust, because it is in ROM
  - It checks the integrity of the next one, before control is passed to it
  - Thus we trust the second, because the first said it's fine & we trust it

  - For every component *i*, component *i-1* must check its integrity
  - We trust *i* if we the check succeeded and we trust *i-1*

  - Thus we can build a Chain of Trust from the Root of Trust



Secure Boot on ESP32

Secure Boot on a microcontroller (ESP32)

- High-level boot flow
  - Boot ROM starts, checks if secure boot enabled
  - Boot ROM authenticates the bootloader
  - If the signature is valid, the bootloader is started
  - The bootloader authenticates the application image
  - If the signature is valid, the application is started

  - If any of the checks fail, the boot process is interrupted

Hardware support – ESP32

- ESP32 contains 4 eFuses (One-Time Programmable memory blocks)
  - 256 bit each, 8 * 32 bit blocks
  - BLK0: used entirely to store system configuration, like if secure boot or firmware encryption enabled
  - BLK1: used to store a symmetric key used by firmware encryption
    - only accessible to HW
  - BLK2: used to store the SHA256 hash of the public key used by secure boot
  - BLK3: available for the application

Detailed Boot Flow 1 – ESP32

- On start-up, ROM code checks if the secure boot is enabled in BLK0

- If enabled, the Signature Block of the bootloader is checked
  - Magic byte & CRC checked

- If the Signature Block is valid, the bootloader image is checked
  - The hash of the key in the Signature Block is compared to the hash stored in BLK2
  - The image is hashed and the hash is compared to the one stored in the Signature Block
  - If the hash is correct, the signature is checked

- If all checks pass, the bootloader is loaded into memory & executed

Detailed Boot Flow 2 – ESP32

- The bootloader checks the Signature Block of the application image
  - Magic byte & CRC checked
- If the Signature Block is valid, the app image is checked
  - The hash of the key in the Signature Block is compared to the hash stored in BLK2
  - The image is hashed and the hash is compared to the one stored in the Signature Block
  - If the hash is correct, the signature is checked
- If all checks pass, the application is loaded into memory & executed
  - The bootloader may check multiple app images, until a valid one is found

UNIFIED EXTENSIBLE
FIRMWARE INTERFACE

UEFI

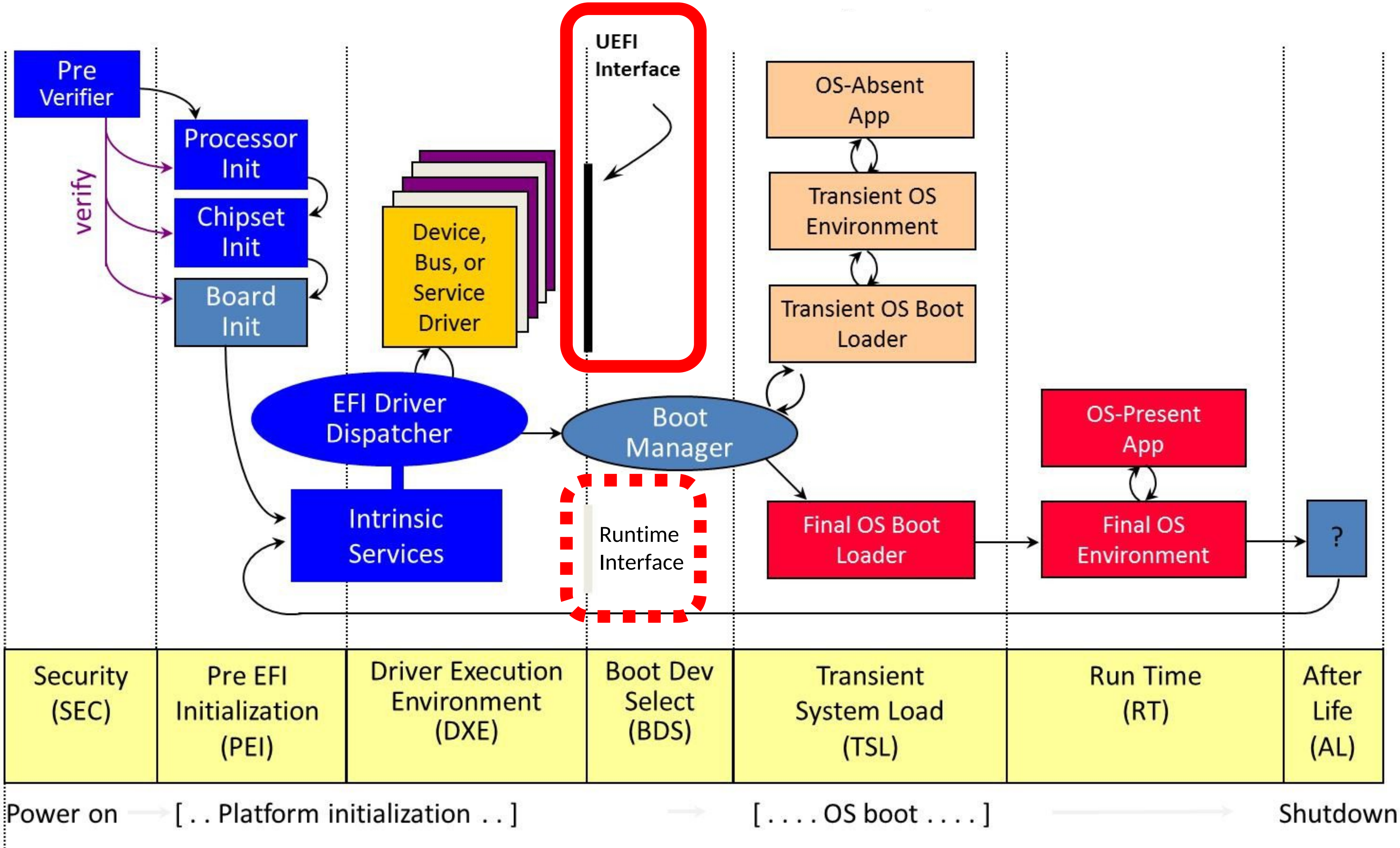
UEFI - History

- A long time ago, in a galaxy far, far away…
  - … there was BIOS (1975)
    - » Basic Input/Output System
    - » Developed by a handful of companies
      - Proprietary software
      - 0 compatibility between the different implementations
    - » Simple boot flow
      - Hardware check
      - Start whatever is at the first sector of the first disk
    - » Secure boot not supported
    - » Some serious limitations
      - Must run in 16-bit mode
      - 1 MB of memory for code
      - Cannot boot drives larger then 2 TB
    - » Intel, mid-1990s: „We need something better"

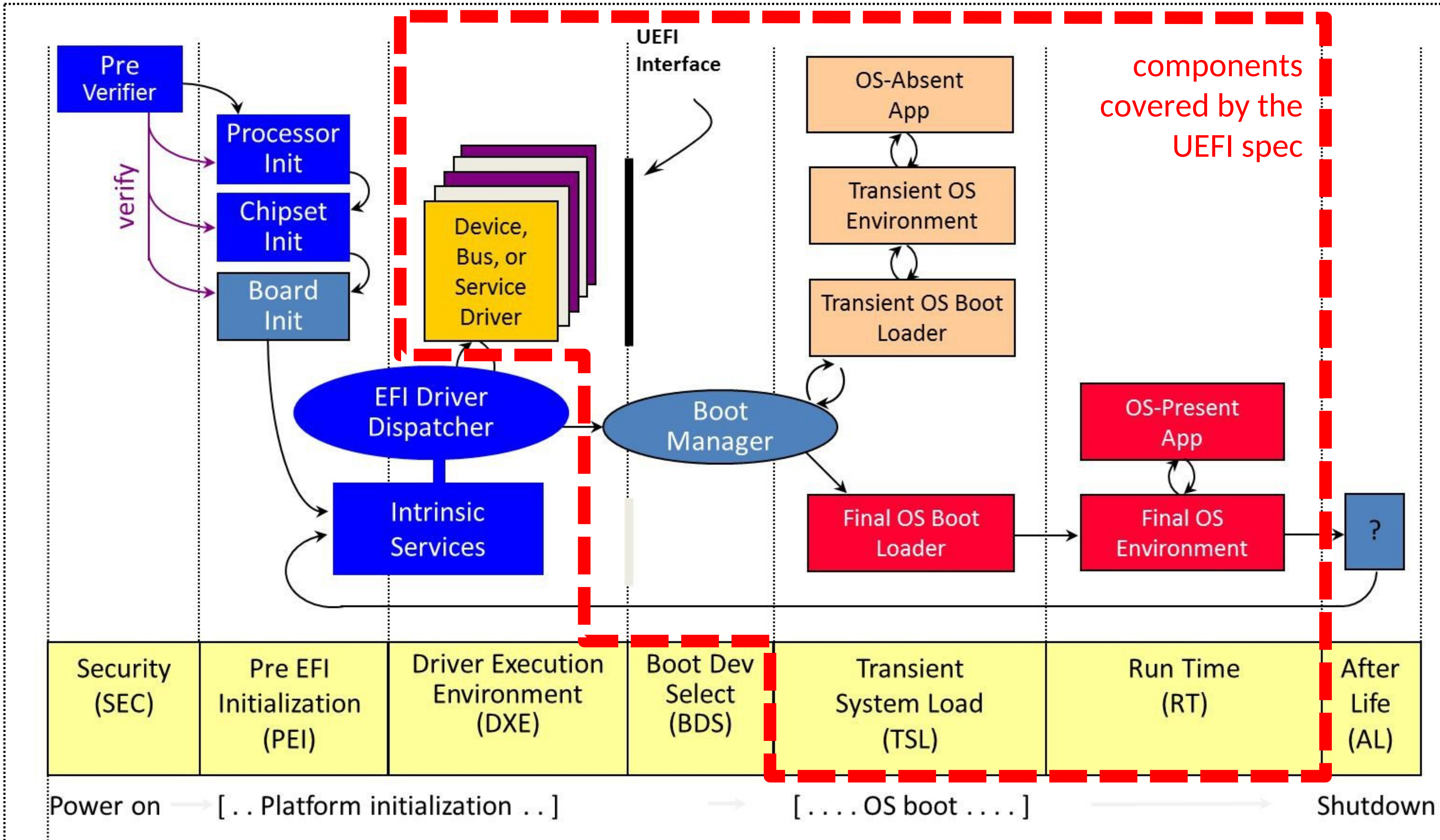
UEFI - History

- 1998 – Intel Boot Initiative
  - Later renamed to EFI
- 2004 – first open source UEFI implementation (Tiano by Intel)
- 2005 – EFI 1.10 were given to Unified EFI Forum
  - An alliance between tech companies to coordinate the specification of UEFI
    - » AMD, ARM, Apple, HP, Intel, Lenovo, Microsoft, etc.
  - Original EFI spec. owned by Intel, UEFI spec. owned by UEFI Forum
- 2006 Jan. – UEFI v2.0 with crypto & security features
- 2007 Jan. – UEFI v2.1 with network authentication & UI
- 2022 Aug. – Latest UEFI spec., v2.10

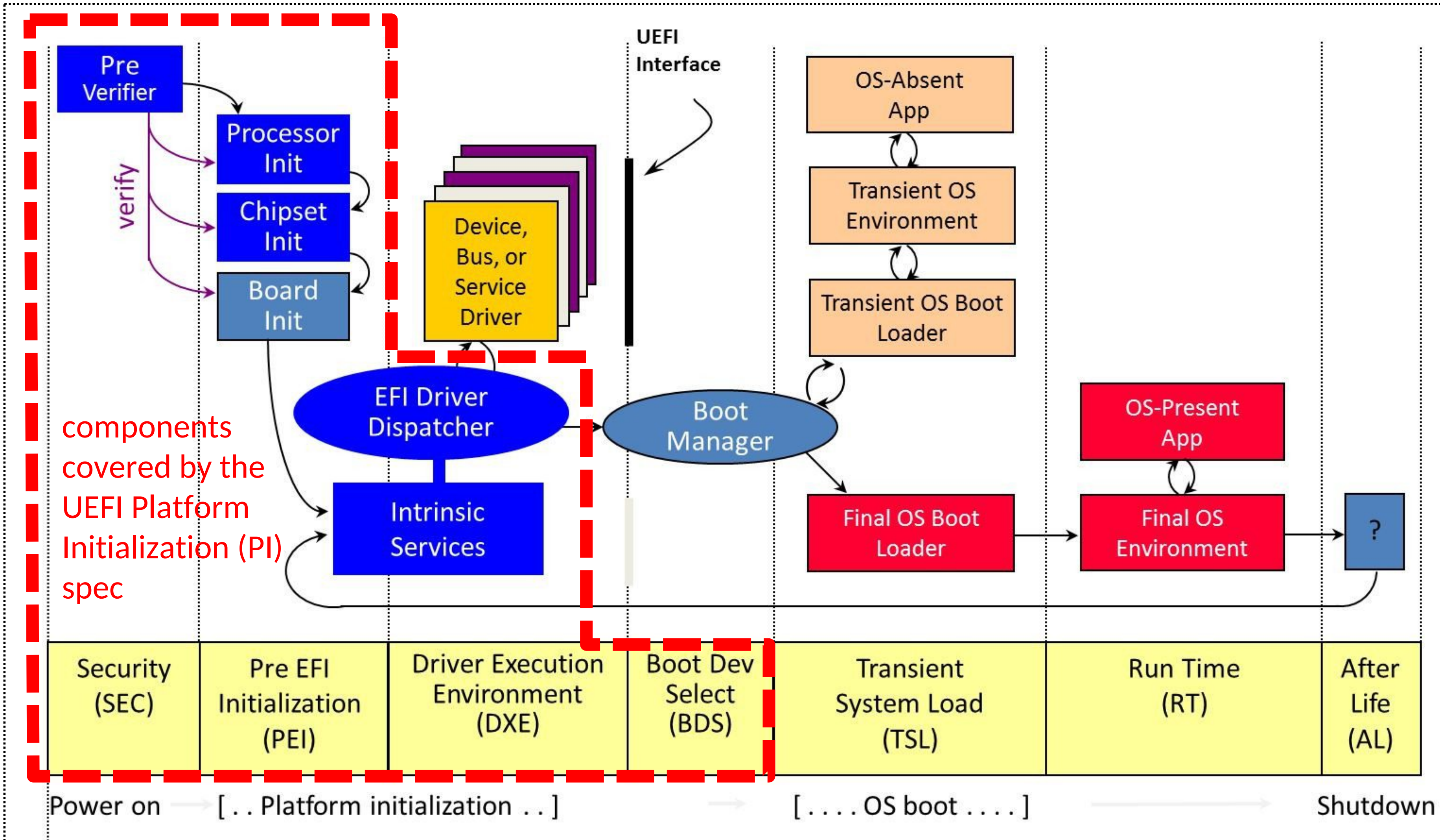
Platform boot flow overview



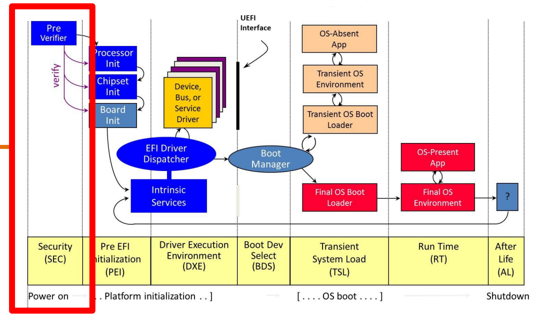
Platform boot flow overview



Platform boot flow overview



UEFI boot process – SEC

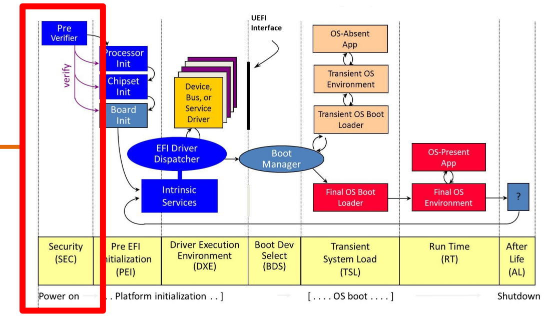


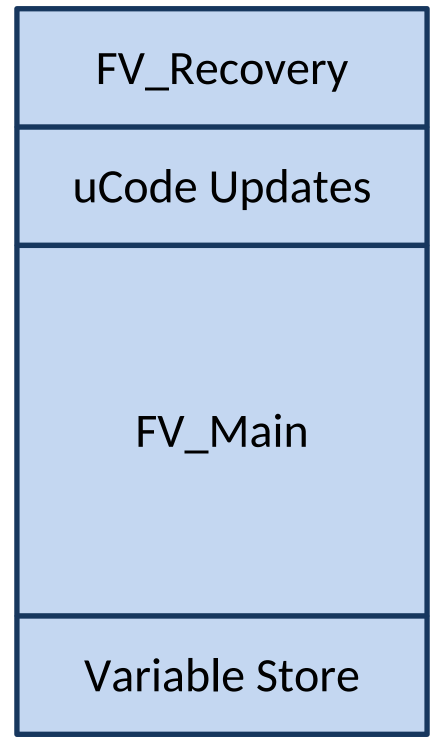
- Very first code executed by the CPU
  - Minimal code fetched directly from an SPI flash
  - Typically hand-coded assembly
  - Architecturally dependent and not portable
- SPI flash stuff
  - The CPU fetches the first instruction from address 0xFFFFFFF0 (re-directed to the flash by hardware)
  - This is just a JMP instruction to the start of the platform initialization code in the flash
- The SEC phase is responsible for
  - Handling all platform reset events (power-on, wake-up)
  - Executing microcode patch update to the CPU
  - Configuring the CPU Cache as RAM (CAR)

    Note that the RAM has not yet been configured, but firmware code in later phases needs a C like execution environment (e.g., a stack to support function calls)
    --» we need some sort of RAM

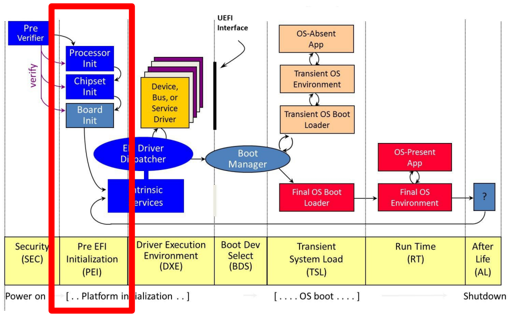
SPI flash content

- Firmware Volumes (FVs)
  - Logical firmware devices
  - Different boot phase code (e.g., PEI, DXE) may be stored on different volumes
- Each FV is organized into a Firmware File System (FFS)
  - A FFS contains files and their meta-data
  - Executable files can be in PE (Portable Executable) or TE (Terse Executable) format
- FV_Recovery
  - Contains the Boot Block, which holds the SEC and PEI phase code
- Other FVs contain compressed UEFI drivers (DXE phase) and remaining UEFI code



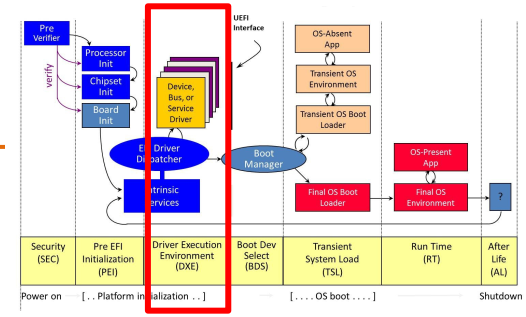


UEFI boot process – PEI



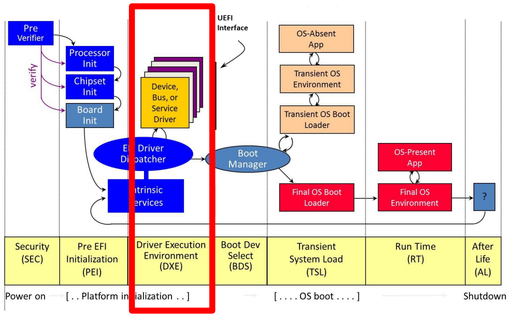
- Control is transferred to PEI phase code
  - PEI = Pre-EFI Initialization
  - Still fetched directly from the SPI flash
- PEI Dispatcher invokes PEI Modules (PEIMs) that perform early hardware and memory initialization (CPU, chipset, board init)
- Last PEIM called is DXE IPL (Initial Program Load), which decompresses FV_Main into main memory (RAM) and transitions to the DXE phase

UEFI boot process – DXE



- Control is transferred to DXE phase code
  - DXE = Driver Execution Environment
  - Code is executed from RAM
- DXE Dispatcher dispatches DXE and Runtime (RT) drivers from FV_Main (in RAM) into main memory (in RAM)
- The DXE phase is responsible for
  - Additional hardware initialization and configuration performed by DXE drivers
  - System Management Mode (SMM) setup
  - Secure Boot enforcement
  - Firmware update signature checks

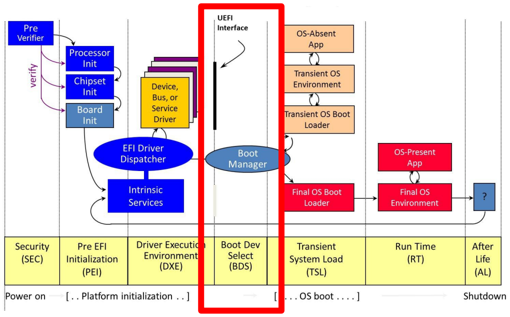
Setting up SMM and RT services



- SMM:
  - A platform-specific driver configures SMRAM and launches the SMM Dispatcher
  - The SMM Dispatcher loads SMM drivers from FV_Main into main memory and executes them
  - Some SMM drivers install SMI interrupt handlers
  - Typically performs tasks like power management and hardware control
  - Should only be used by the firmware
  - Transparent to the OS (Ring -2)

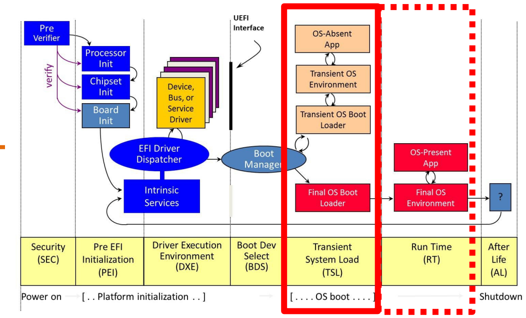
- RT services:
  - Some RT drivers install services callable by the OS at runtime
  - Note that DXE drivers are unloaded after OS boot, leaving only RT and SMM drivers to be available at runtime

UEFI boot process – BDS



- Control is transferred to BDS phase code
  - BDS = Boot Device Selection
  - Typically, the BDS phase code is encapsulated in a single file loaded by the DXE phase
- The Boot Manager consults configuration information to decide where the OS should be booted from
- It has access via the UEFI interface to all UEFI Boot Services that the DXE phase set up --» it can use them to access the file system on the hard drive in order to find an OS bootloader
- If UEFI Secure Boot is turned on, it also checks the integrity of the OS bootloader before starting it

UEFI boot process – TSL (and RT)

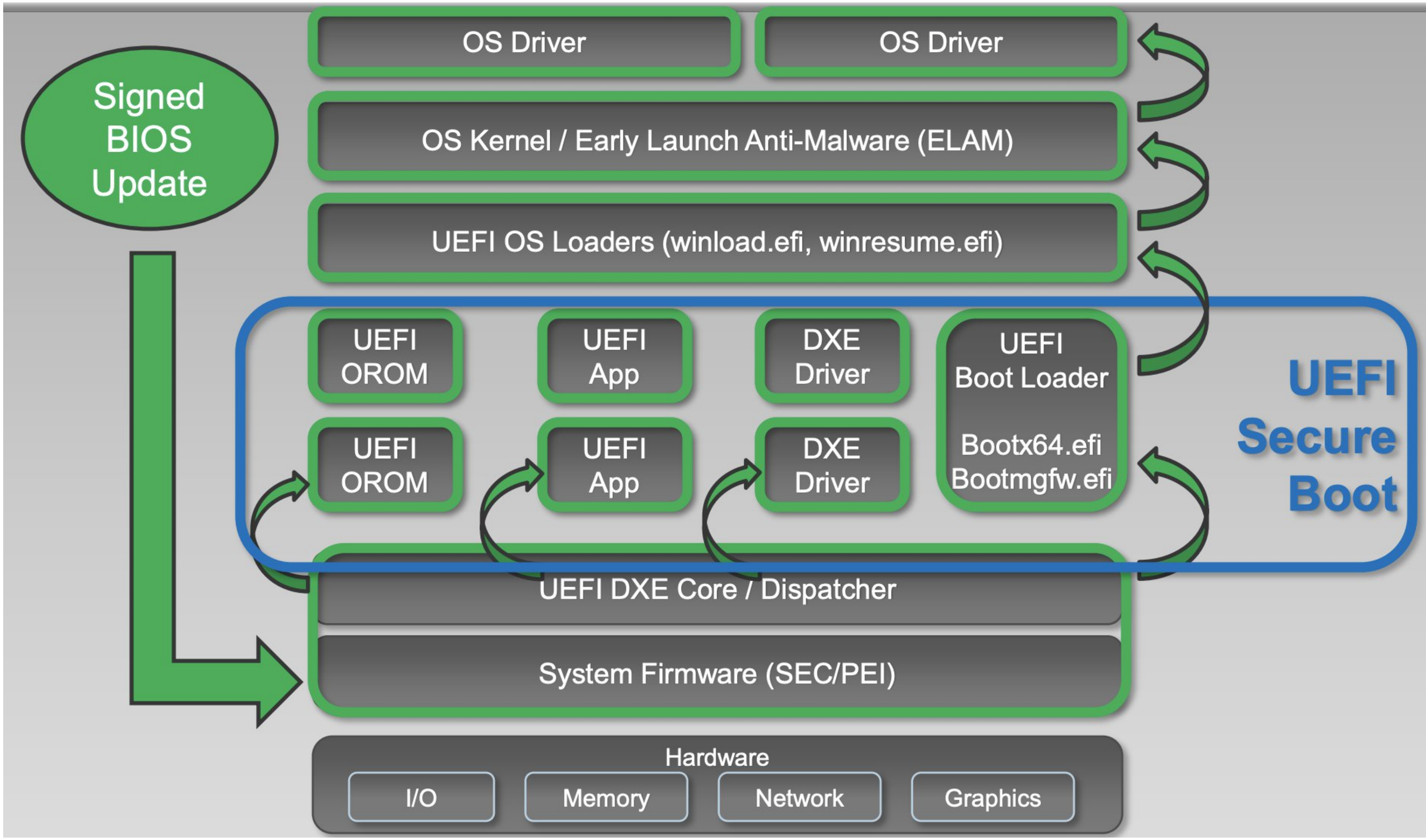


- Control is transferred to TSL phase code
  - TSL = Transient System Load
  - This is typically the OS bootloader loaded from the SSD/HDD
- The OS bootloader loads the OS kernel into memory
- It can still use the Boot Services, set up in earlier phases, available via the UEFI Interface
- Before passing control to the OS kernel, it calls ExitBootServices() via the UEFI Interface
  - Memory holding DXE drivers is freed up
  - Only RT drivers (and SMM code) remain resident and can be used by the OS as runtime firmware services available via the Runtime Interface (or via SMIs)

UEFI Secure Boot

- General idea:
  - Verify if an executable (e.g., OS bootloader) is permitted to load and execute during the UEFI boot process
  - Verification is based on checking digital signatures on code (or checking code hashes)
    - » if executable is signed with an authorized key --» allow
    - » if hash of executable is stored in DB of authorized hashes --» allow
- Caveats:
  - Secure Boot is optional; the way it is managed, enabled or disabled is a decision of the platform manufacturer and the system owner
  - Security boils down to managing authorized keys and hash databases
  - Flash based UEFI components (SEC, PEI, DXE Core) are not verified
    - » they are implicitly trusted
    - » the flash image itself is signed, but it is verified only when loaded into the flash during a firmware update --» see UEFI secure firmware update later…

UEFI Secure Boot – Chain of Trust



http://c7zero.info/stuff/Windows8SecureBoot_Bulygin-Furtak-Bazhniuk_BHUSA2013.pdf

UEFI Secure Boot – Verification process

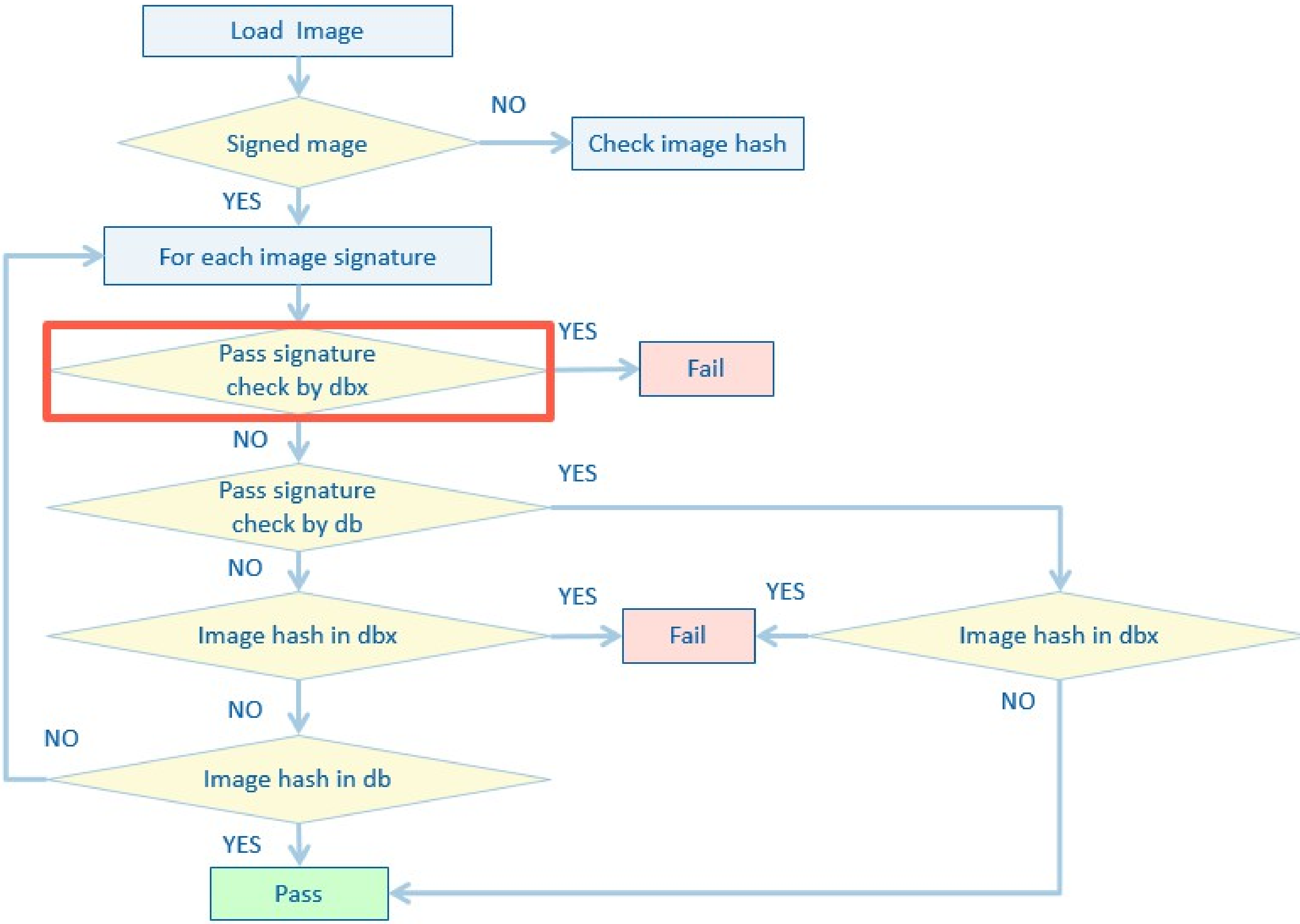
- Signature and hash checks
  - db file: trusted certificates and hashes
  - dbx file: revoked certificates and untrusted hashes

- The db files are stored in the SPI flash and later copied to memory

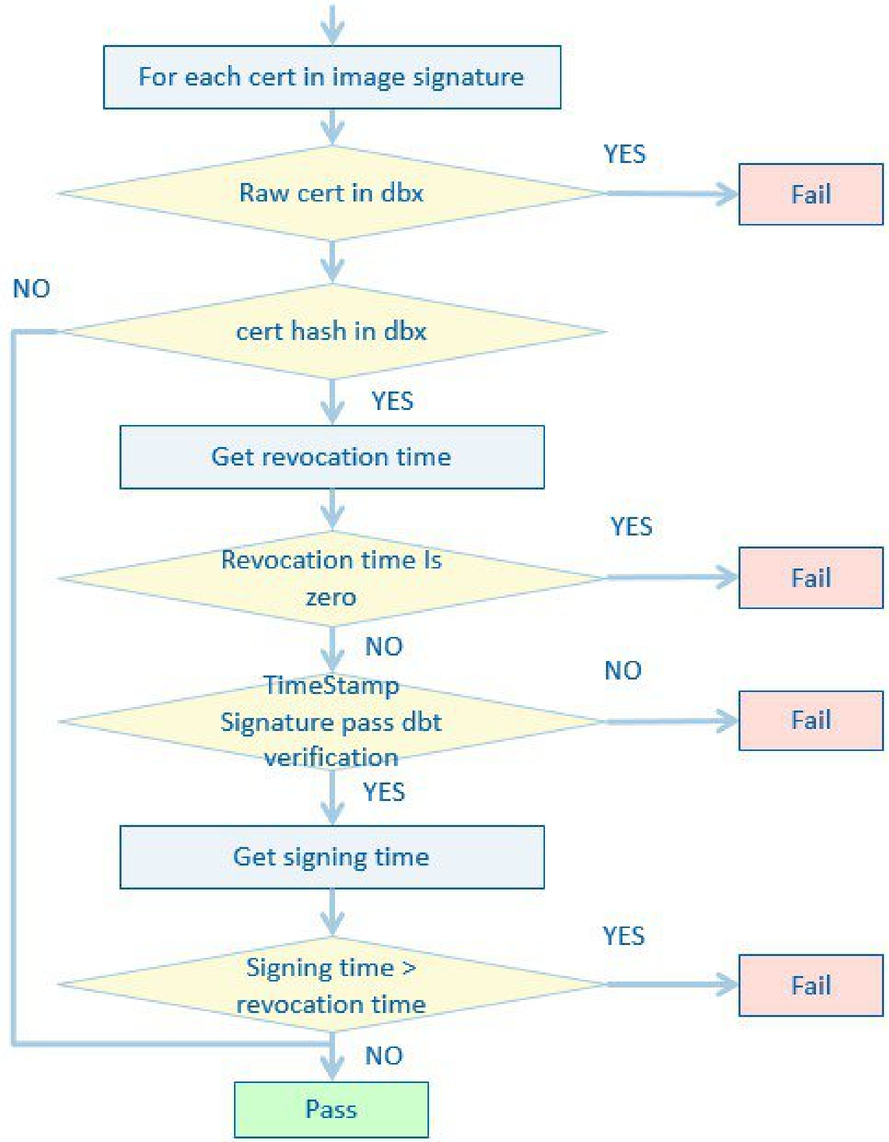
- Instead of adding a lot a signer certificates to the db, developers can ask Microsoft a certificate signed by their CA, which can be added to the db

- Linux distributions were also signed by developers’ keys signed by the Microsoft CA

UEFI Secure Boot – Verification process



UEFI Secure Boot – Verification process



Secure Firmware Update

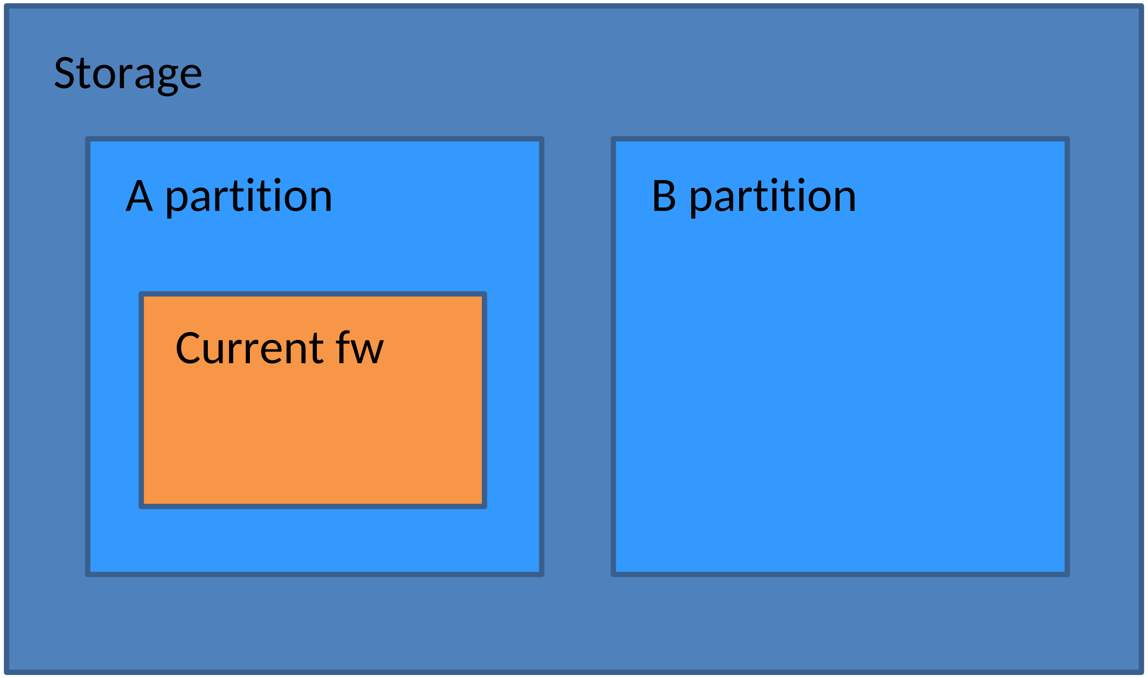
Secure Firmware Update

- Software contains bugs & vulnerabilities
  - Devs fix bugs, new version must be deployed somehow
  - Usual update processes work on a package granularity
    - » Some software components cannot be bundled to packages
- Downloaded components must be integrity-checked
  - Just like in the case of secure boot
- The update mechanism must be fail-safe
  - The system must be able to recover in case of faults
  - Rollback must be possible – use the previous version, if the new cannot be used for some reason
- Rollback protection needed
  - An attacker must not be able to install an older version
    - » Version numbers must be checked & they must be integrity protected as well

A/B partitioning

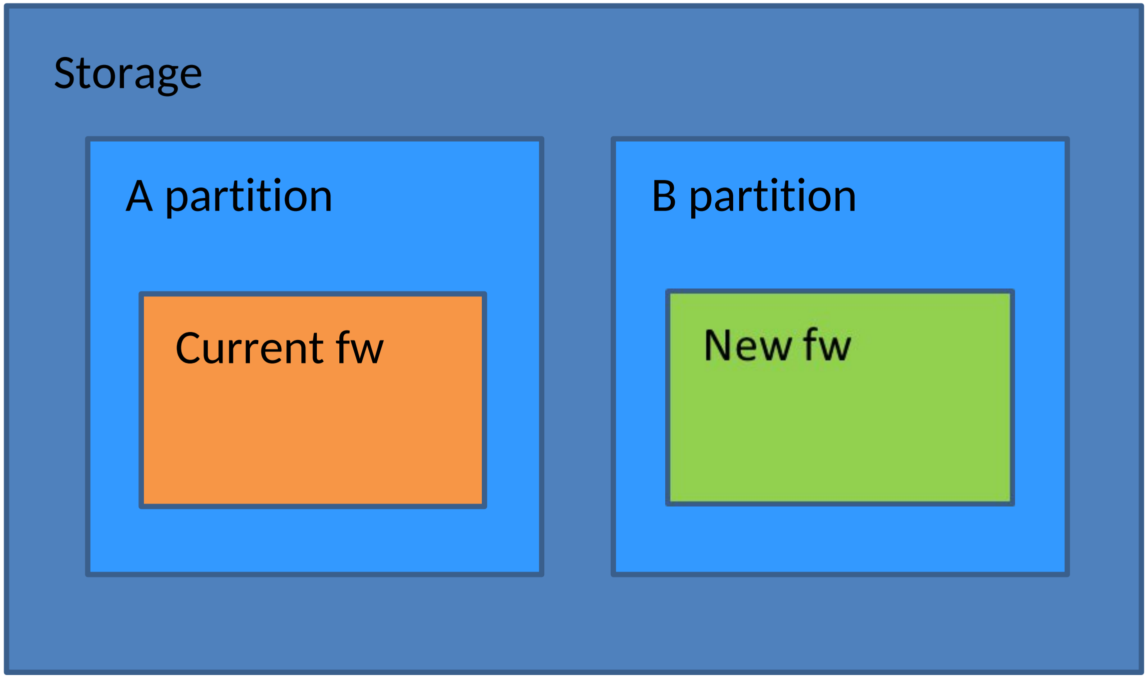
A/B partitioning

- Current firmware downloads new version to free partition



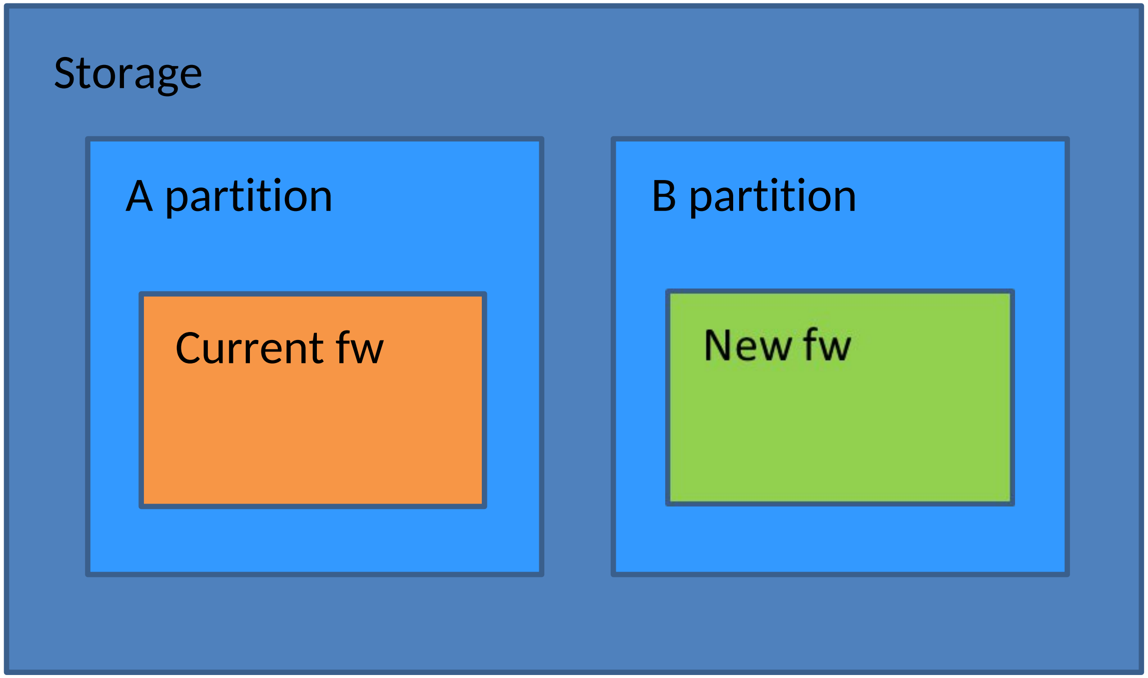
A/B partitioning

- Current firmware downloads new version to free partition



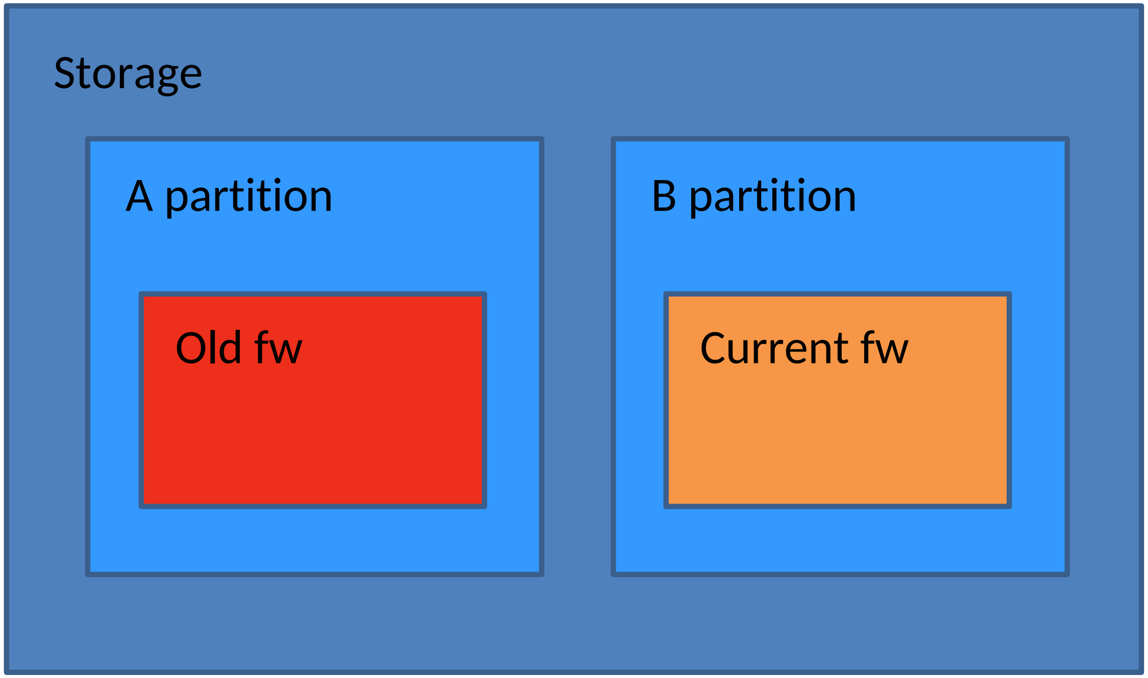
A/B partitioning

- Current firmware downloads new version to free partition
- Device reset, boot into new firmware



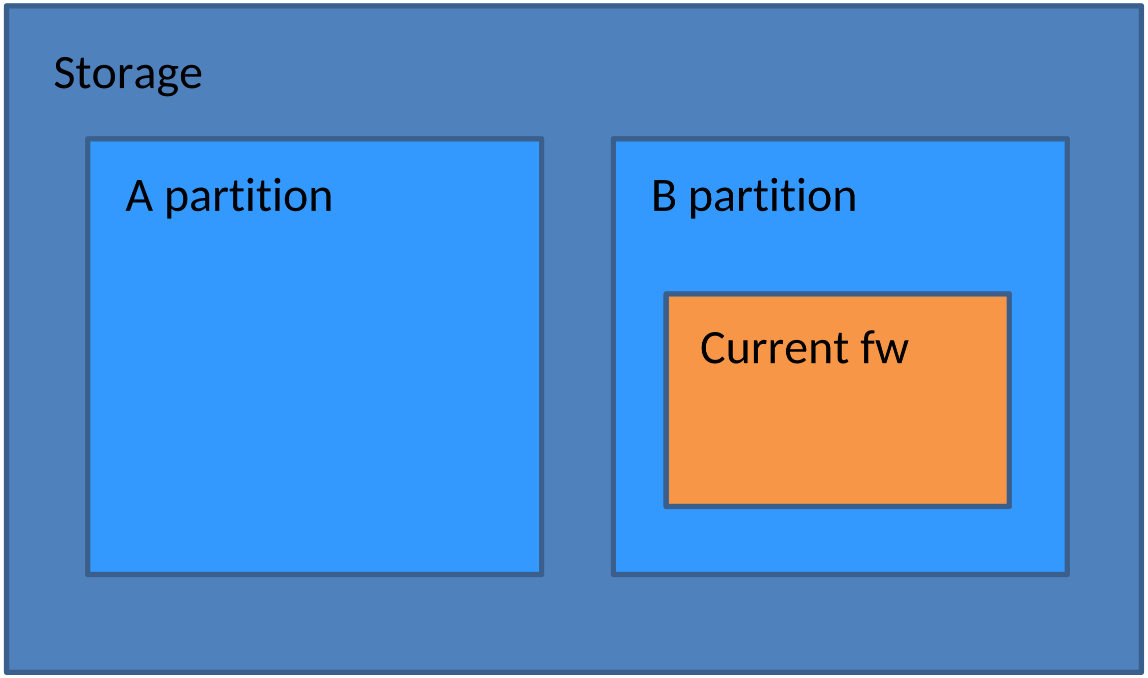
A/B partitioning

- Current firmware downloads new version to free partition
- Device reset, boot into new firmware
  - If successful, old firmware can be deleted



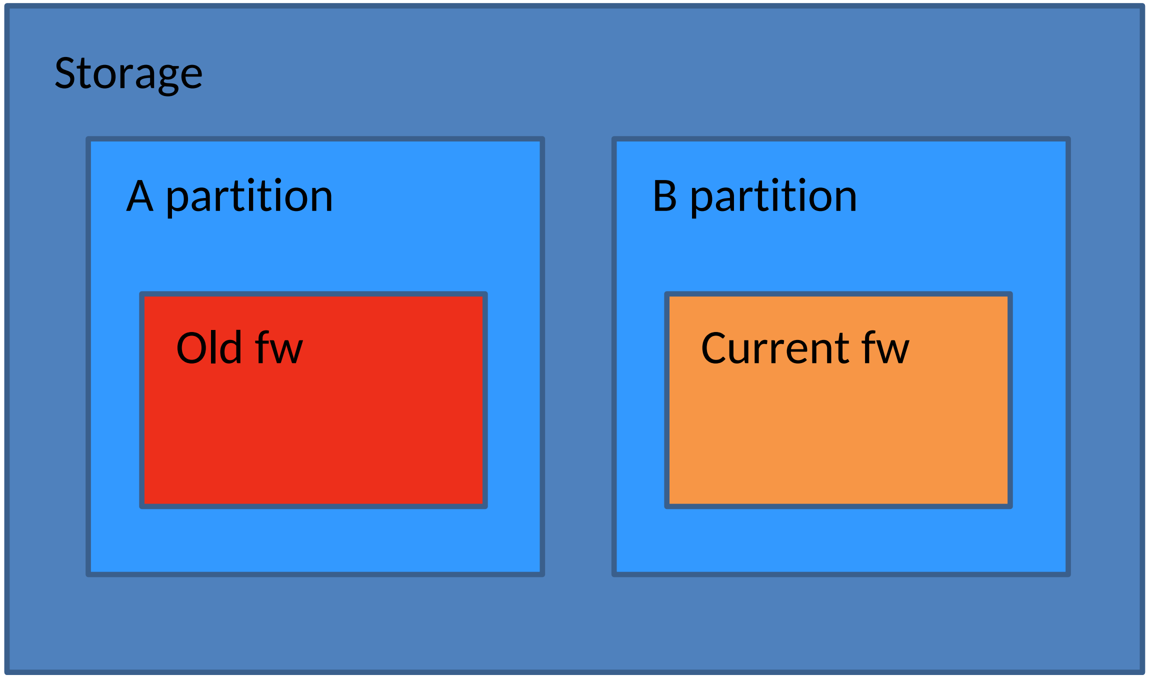
A/B partitioning

- Current firmware downloads new version to free partition
- Device reset, boot into new firmware
  - If successful, old firmware can be deleted



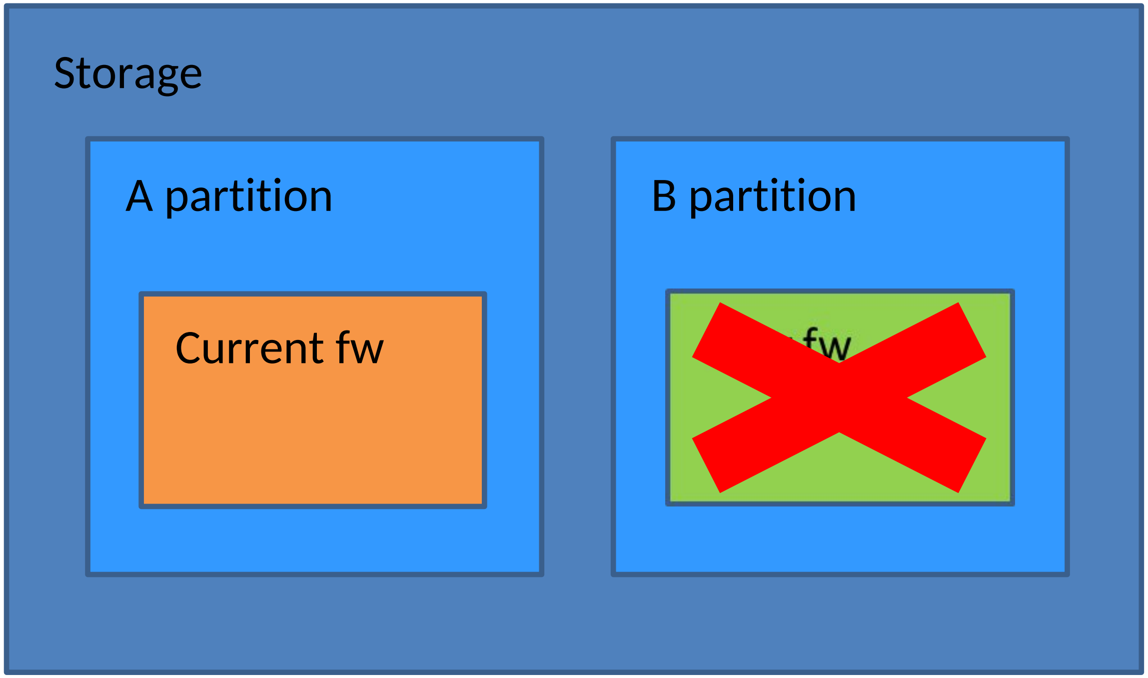
A/B partitioning

- Current firmware downloads new version to free partition
- Device reset, boot into new firmware
  - If successful, old firmware can be deleted
  - If not, rollback to original



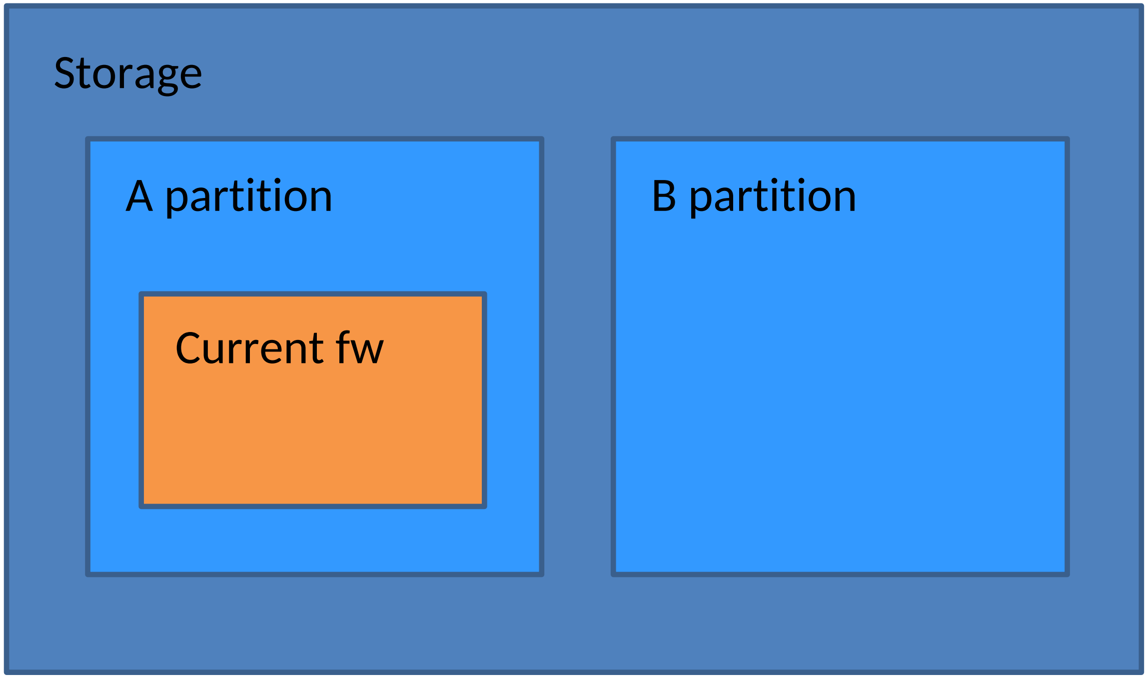
A/B partitioning

- Current firmware downloads new version to free partition
- Device reset, boot into new firmware
  - If successful, old firmware can be deleted
  - If not, rollback to original



A/B partitioning

- Current firmware downloads new version to free partition
- Device reset, boot into new firmware
  - If successful, old firmware can be deleted
  - If not, rollback to original



ESP32 Secure Firmware Update

ESP32 Over The Air (OTA) update

- The device has a partition table that describes the content of the flash
  - Required partitions for OTA update
    - ota_0, ota_1, to store downloaded images
    - otadata , to store data about the update process (e.g. which partition to use on next boot)
- Some functions are implemented, but the developer must perform:
  - Checking periodically for new images
  - Download & write the new firmware
    - Preferably by using a secure channel
  - Check the integrity of the downloaded image
    - Supported: SHA256 hash
  - Perform self test & cancel rollback in the new image

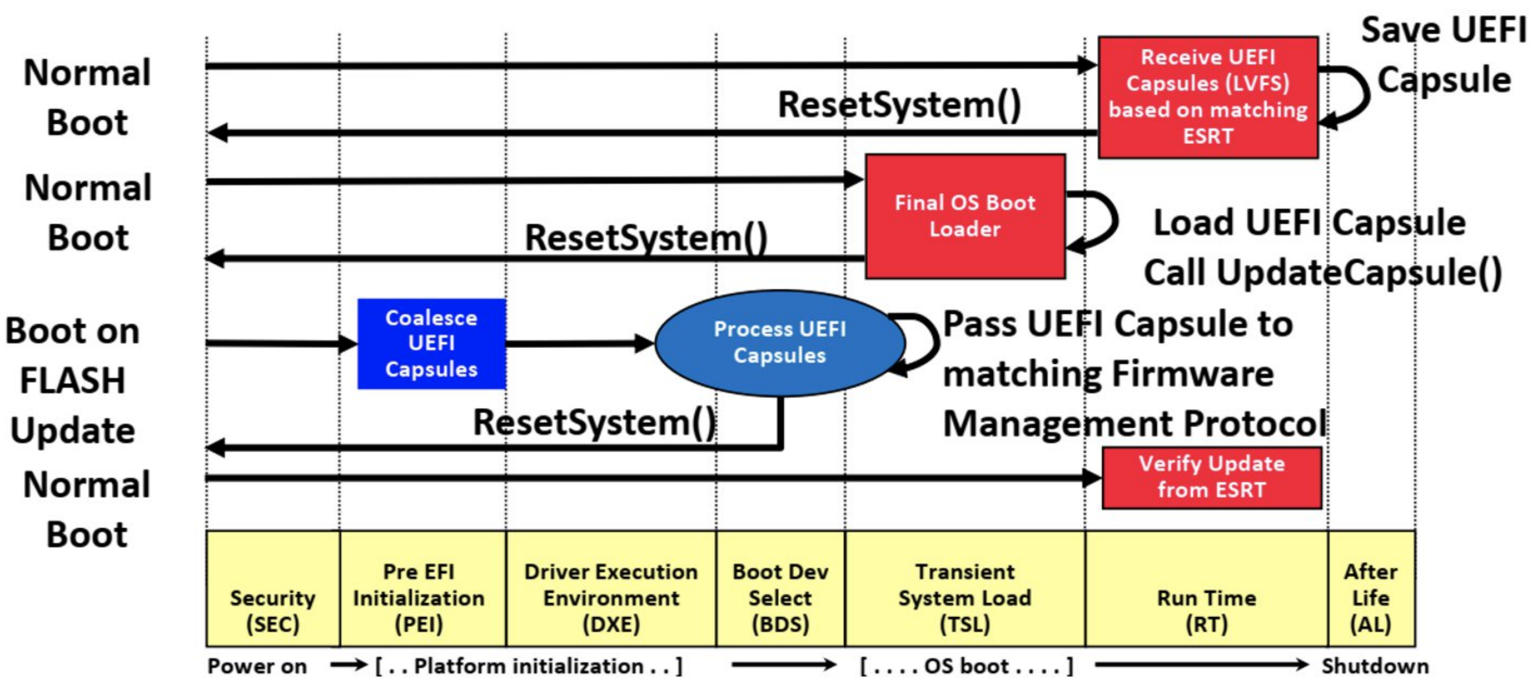
ESP32 Rollback Protection

- All firmware images include a secure version number
  - Must be incremented by the developer before release
  - As it's part of the image, it cannot be changed without changing the hash
- When the partition containing the new image is marked to be executed on next boot, this version is checked
  - If smaller than the current, the partition is erased
- The bootloader checks the version before passing execution to the firmware
  - The partition is not selected for execution, unless the version is greater than or equal to the current version

UEFI Secure Firmware Update

UEFI Secure Capsule Update

- Capsule: a header + data, signed by a key known to the current firmware
- It must be downloaded and copied to a partition accessible to UEFI



UEFI Secure Capsule Update

- HW protection
  - Only the firmware itself can write the SPI flash
- Recovery
  - During update, it is possible to save the content of the SPI flash to disk, to support automatic recovery, in case the freshly downloaded firmware version fails to boot properly
- Rollback protection
  - UEFI firmware contains a table for configuration-related information
    - » EFI System Resource Table (ESRT)
  - The ESRT contains a Lowest Supported Firmware Version field
  - In case of a failure, rollback is possible, but not beyond the Lowest Supported Firmware Version
    - » Rollback to versions with known vulnerabilities can be restricted

Key takeaways

- Secure boot
  - Requires hardware support
  - Each component is checked by its predecessor
  - For the first steps, check what the vendor recommends

- Secure Firmware update
  - Needs to be fail safe
    - Self test needed
    - The previous version must be available
  - Must be protected from rollback attacks
    - Version numbers must be checked & protected

Control questions

- What are the goals of a platform firmware?
- Why is firmware security important?
- What are the phases of the UEFI boot process?
- How is the Root of Trust provided for ESP32?
- What does UEFI stand for? What is standardized by the UEFI Forum?
- How are images verified by UEFI Secure Boot before loading and executing them?
- What keys and databases are used by UEFI Secure Boot? How are they managed?
- What is the goal & the main idea behind A/B partitioning?
- Why self testing is important?
- How can we prevent version rollback attacks?
- What is UEFI Secure Capsule Update?