Secure Boot, Chain of Trust and Data Protection

Akshay Bhat
Technical Director - Security, Timesys Corporation
1905 Boulevard of the allies, Pittsburgh PA 15237, United States
akshay.bhat@timesys.com

Abstract — Secure boot is designed to protect a system against malicious code by ensuring only authenticated software runs on the device. Secure boot can be achieved by verifying digital signatures of the software prior to executing that code. This paper explores secure boot techniques extending the authentication scheme all the way from bootloader to userspace, using open source solutions. There are a plethora of options available and it takes considerable time understanding the advantages and limitations of each solution. This paper aims to reduce the learning curve by giving an overview of each solution.

In addition to software authentication, protecting user data and intellectual property can be critical in certain applications, which can be achieved by means of encryption. This paper also presents various hardware assisted options for secure key storage and concludes with additional considerations in order to implement end to end device security.

I. INTRODUCTION

In case of microprocessors, the first piece of software that runs is the bootloader that resides in read-only memory (ROM). The ROM bootloader is flashed into the microprocessor by the device manufacturer and is responsible for loading the next piece of software which typically is a user bootloader (e.g: u-boot, barebox, little kernel etc) from an external storage media. In-order to achieve secure boot, ROM bootloader needs to support verifying the authenticity of the next stage bootloader before executing it. Most modern microprocessors now support secure boot.

In the microcontroller world, there is no ROM bootloader code built-in to the microcontroller. So to establish secure boot, one needs to ensure the first piece of code that runs on the microcontroller is stored in read-only-memory that cannot be overwritten. Essentially this translates to locking the flash memory region from further programming. This piece of software should implement the digital signature check of the next piece of software. Open source cryptographic libraries such as mbed TLS is light enough to perform the signature verification in microcontrollers.

We shall concentrate on ARM based microprocessors / system on chips (SoC) in this paper. The high level mechanism to authenticate software involves the below steps:

- Create a public/private key pair
- Compute the hash of the software image
- Sign the hash with private key using signing tools (vendor specific or open source)
- Append the signature/certificate along with public key to the bootloader image
- Flash the public key (or hash of public key) onto One-Time programmable (OTP) fuse on the processor

The processor ROM bootloader on power-up loads the user bootloader along with the signature/certificate appended to it. It then verifies the software by performing the following steps:

- Verify the hash of public key appended to bootloader image matches the one stored in OTP fuse
- Extract the hash of bootloader from the signature using the verified public key
- Compare the extracted hash with the computed hash of the bootloader. If it matches it proceed with the boot process, thus authenticating the software

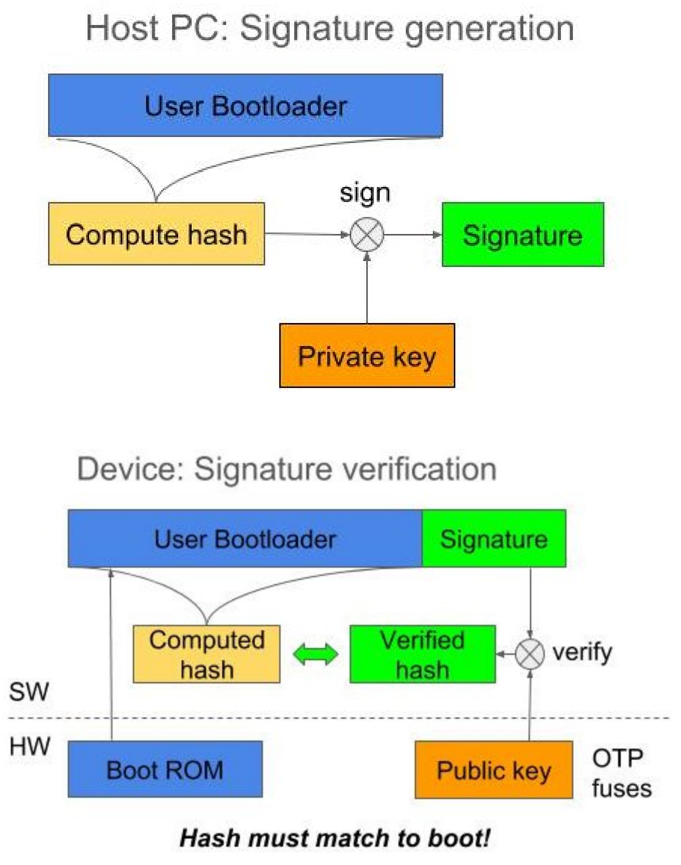

Fig. 1. *Bootloader Authentication*

II. Chain of Trust

In case of a Linux based operating system, there are multiple software components that are loaded and executed during runtime. Below is a list of typical components:

- Bootloader
- Kernel, Device Tree
- Root Filesystem + User applications
- Application data
- Optional: Firmware (FPGA, co-processor), Secure OS (op-tee, arm-trusted-firmware)

In-order to secure the product, the software authentication scheme must be extended all the way to user space, establishing a chain of trust. i.e. ROM verifies signed bootloader, bootloader verifies signed kernel and kernel verifies/mounts encrypted/signed root filesystem (RFS).

The two common approaches to authenticating the kernel as below:

SoC specific mechanism

Similar to how the ROM bootloader authenticates the user bootloader, the scheme can be extended to authenticate the kernel. A combined zImage (kernel + device tree + ramfs) is signed using vendor tools. The bootloader is configured to verify the signature of zImage using with the public keys present in the OTP fuses.

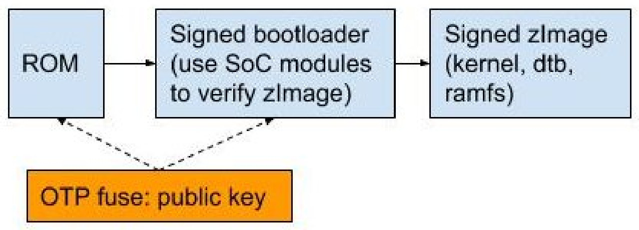
ROM
Signed bootloader (use SoC modules to verify zImage)
Signed zImage (kernel, dtb, ramfs)
OTP fuse: public key

Fig. 2. Extending Chain of Trust using SoC

FIT image

FIT stands for Flattened Image Tree and is a single binary that can contain multiple images with metadata for storing image information along with signatures, thus making it convenient to authenticate. A typical secure boot use case is to generate a FIT image containing kernel, device tree and initramfs. The FIT image is then signed using a private key, and the signature is embedded inside the FIT image. The public key is then embedded inside bootloader. Since the signed bootloader is authenticated by the ROM, we can trust the public key inside of bootloader to verify the FIT image.

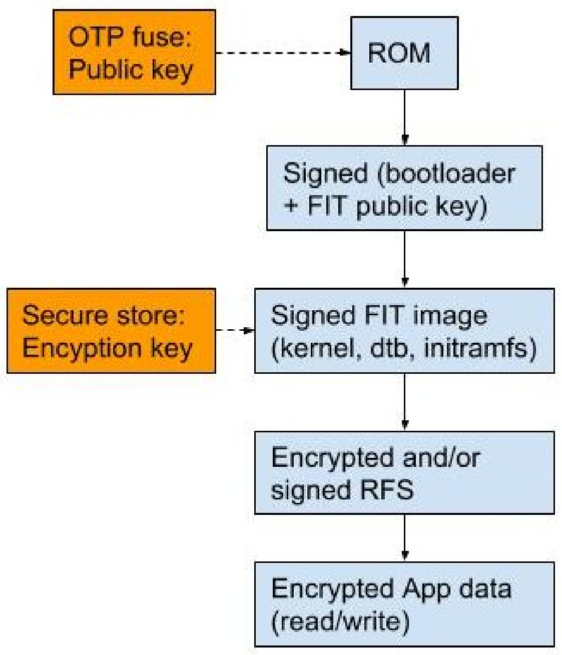
OTP fuse: Public key
ROM
Signed (bootloader + FIT public key)
Secure store: Encyption key
Signed FIT image (kernel, dtb, initramfs)
Encrypted and/or signed RFS
Encrypted App data (read/write)

Fig. 3. Extending Chain of Trust using FIT image

III. Root Filesystem and Application Data Protection

If a root filesystem is read only and small enough to run out of RAM, then embedding the root filesystem inside the FIT image or zImage should be sufficient for authenticating it. However, most times root filesystems are large, and we need to leverage on the mechanisms provided by the Linux kernel for authenticating and/or encrypting its contents.

While secure boot ensures authenticity, it does not protect the device from being counterfeited or prevent hackers from extracting user/application data from the device offline (i.e. reading the non-volatile storage like NAND, eMMC using external hardware mechanisms). If data confidentiality and/or anti-counterfeit functionality is needed, then software/user data needs to be encrypted.

In Linux based embedded systems, encryption of user space software/data can be done at block layer (disk/partition level) or filesystem layer.

Block devices

Full disk encryption and/or authentication involves encrypting and/or verifying the contents of the entire disk at a block level. This is performed by the Linux kernel's device mapper (dm) modules. This method can be used with block devices only (eg: EMMC, SSD, SD card). The authentication and encryption happen on a per block basis at runtime only when that block is accessed, thus avoiding any delays during device boot. For full disk encryption or verification, we need an initramfs (initial RAM filesystem) which is a minimal filesystem responsible for unlocking/mounting an encrypted RFS partition and verifying the signature of the signed RFS.

Depending on whether encryption or authentication or both are required, the below modules can be used:

- dm-crypt [1](encryption)
- dm-verity[2] (authentication - read-only filesystem)
- dm-integrity[3] (can be used in conjunction with dm-crypt for encryption and authentication - read/write filesystems)

Filesystems

An alternate approach is to encrypt and authenticate on a file or directory basis. Below are some mechanisms to achieve the same:

- fscrypt (ext4, ubifs etc) - integrated into filesystems
- ecryptfs - encrypted filesystem stacked on-top of filesystem

IV. Secure Key storage

On a desktop or cell phone, the key used to encrypt the filesystem is derived from a user password entered interactively. Embedded devices typically do not have this luxury. Hence we need to store and protect the key on the device. Below are some protection mechanisms:

Secure key storage built-in to SoC

Most modern SoC make provisions for secure key storage. It is either in the form of protected eFuses or a cryptographic module that can encrypt user keys using a unique master key pre-programmed onto the processor. SoC based mechanism would be the preferred route since there is no additional cost associated and the key is never exposed outside the processor. Depending on the processor capabilities it can be further secured using tamper detect techniques and memory encryption.

Trusted Platform Module (TPM)

The keys can be stored on external chip such as TPM and then sealed using the Platform Configuration Registers (PCR) registers. The fundamental concept behind sealing is to measure the state of software running on the microprocessor such that a modification to the software will result in the unseal process to fail, thereby rendering the encryption key being inaccessible. However one should be wary of external security chips such as TPM being susceptible to I2C bus attacks[4].

Secure key storage on a external crypto chip (eg: ATECC508A)

Secure elements such as external crypto chips can be leveraged to store the filesystem key externally. However on a headless device these chips suffer from the same fundamental issue of requiring the software running on the microprocessor to authenticate with the external crypto chip in order to retrieve the key. This would in-turn result in requiring a protection mechanism for the authentication key defeating the purpose of the secure element.

Trusted Execution Environment

ARM processors support TrustZone, running a secure OS in a Trusted Execution Environment (TEE) [5] which can be used to implement a software based secure key storage. The Linux kernel can then avail services from the secure OS to store/retrieve keys.

V. Additional Security Measures

Apart from securely storing keys, in order to ensure end to end product security few other aspects need to be considered:

- Most bootloaders have a associated environmental variable section that contains runtime persistent settings. It is strongly recommended to encrypt and/or authenticate this section of memory.
- For development purposes debug ports such as JTAG and serial console come handy. When moving from development phase to production, the debug ports need to be setup for authentication and/or disabled.
- One needs to consider a key revocation strategy in-case of compromised keys. The keys for higher level software can be updated by designing a secure firmware update mechanism. To revoke keys residing in OTP fuses requires processor support, something to keep in mind during the processor selection.

VI. TRADE OFFS

Implementing security on products can result impact the overall device performance. The most common areas that might require review are:

Boot time

Products requiring quick boot times should be wary of potential delays associated with authenticating images. In our testing we found that authenticating a FIT image without hardware acceleration can add 0.5 to 1s to the boot time, adding a initramfs to mount a encrypted filesystem can add 2-3 seconds to the boot time.

File system performance

Cryptographic operations take considerable CPU cycles and can impact the filesystem read/write performance. It is advisable to choose hardware accelerated cryptographic algorithms where possible.

CONCLUSION

Security cannot be introduced as an afterthought; it needs to be baked into product design. The journey begins from choosing the processor with the required security hardware modules and goes onto layering the various security blocks in software to achieve product security.

REFERENCES

[1] https://gitlab.com/cryptsetup/cryptsetup/wikis/DMCrypt

[2] https://source.android.com/security/verifiedboot/dm-verity

[3] https://gitlab.com/cryptsetup/cryptsetup/wikis/DMIntegrity

[4] https://github.com/nccgroup/TPMGenie

[5] https://www.timesys.com/security/trusted-software-development-op-tee/