Security and Privacy Technologies

July 2021

Secure and Trusted Boot Architecture and Application for Access Networks

Jointly prepared by CableLabs® and COMMSCOPE®

Yuan Tian, Security Engineer | y.tian@cablelabs.com
Massimiliano Pala, Ph.D., PKI Architectures, Director | M.Pala@cablelabs.com
Steve Goeringer, Distinguished Technologist | S.Goeringer@cablelabs.com
Sasha Medvinsky, Engineering Fellow | sasha.medvinsky@commscope.com
Ali Negahdar, Distinguished Software Engineer | ali.negahdar@commscope.com
Tat Chan, Ph.D., Distinguished System Engineer | tat.chan@commscope.com

Executive Summary

How do you know that you can trust a computer system or network device when you turn it on? Whether it is the first time the system is being used or it is just being rebooted, how can you be sure that the system software is safe and secure? Perhaps the system was tampered with while in transit to its point of use, the software was corrupted while it was not connected to the network. Maybe a hacker compromised the software and triggered a remote restart to load the new code. There are many possibilities for reality to differ greatly from expectations. Fortunately, there are technologies that can be applied to ensure that reality is as you expect it—secure boot and trusted boot technologies provide approaches to make it hard for adversaries to compromise the software of deployed systems.

You might be asking why the security of the boot process matters and what aspects must be considered when approaching this technology for our networks and devices. By answering these and other questions related to trusted boot technology, we discuss new opportunities for deploying more secure access networks that are robust against many forms of advanced attacks that can be the source of service theft or even infrastructure damages (e.g., ransomware).

In this paper, we provide an overview of different types of authenticated boot technology and discuss the architecture, principles, and threats associated with different aspects of this technology. Specifically, we look at how the broadband industry can leverage trusted boot to reduce the risk of compromises for its networks. Where applicable, we provide deployment recommendations that are specifically targeted for the cable access network.

Introduction

As our network expands with unprecedented speed to connect more and more endpoints, we need to ensure that this expansion still meets the standards of reliability, availability, and confidentiality. Within cable networks, the secure authentication of endpoints has always been a primary concern. For this reason, the broadband industry has been working hard to deploy provisioning servers, network management systems, and authentication tools to secure endpoints' communication.

All these operations are meaningful only as long as the endpoints can securely access the network. They can, however, be attacked and infected with hard-to-detect malware prior to doing so, which may cause severe damage to endpoints or even spread malware through the entire network and compromise users' privacy.

The first threat example is software component tampering. If the firmware is not protected or locked, the attacker can then modify the firmware directly and inject rootkits to allow remote (or backdoor) control and software eavesdropping. One example of this type of attack is BOOTRASH, which uses unallocated disk space between partitions to store the Nemesis bootkit components.

A second threat to boot security is related to secure software download. If the firmware update process is not authenticated, an attacker could successfully update the firmware with their own compromised image by simply tampering with the configuration of the location of the new firmware to be downloaded. In the broadband use case, for example, an attacker could change the address of the Trivial File Transfer Protocol (TFTP) server that is used to host the Code File images and force the device to connect to a malicious TFTP server, which may be used in conjunction with a modified image for such a device to startup or reset. This attack can be performed locally and remotely.

Another threat associated with malicious code execution is runtime code injection like the ZeroAccess attack. In this process, the attacker first installs malicious code on their website and tricks a victim into visiting it. Once a victim clicks on the attacker's HTTP link, the malicious PHP script will be automatically downloaded, proceeding to install malicious code that enables the escalation of privileges and bypasses security controls to access critical assets. If secrets are embedded in firmware, the attacker may reverse engineer the firmware content and discover the secret.

The conclusion here is obvious: in addition to network management over secure channels, we need to provide a mechanism for protecting and ensuring the integrity of all the endpoints before they join the network and while they are operational. That is where we start to think about boot security, as booting is the starting point for every device in the cable network.

Because the foundation of a secure platform starts with the authenticity of the software that runs on that system, boot-up code validation starts at the very beginning of the boot process in the Boot ROM (usually a trusted component) and continues with each stage of the boot up, such as bootloader, Kernel, and RFS, until the entire runtime code is validated. If the validation of the code fails at any stage, the platform aborts the boot up from the invalid image; it normally initiates a recovery procedure when backup images are available. Practically, the secure boot process prevents the execution of unauthorized code on the platform and provides a mechanism on the target device to validate subsequent components. Lack of a secure boot allows bad actors to inject rogue code into the code execution path and take control of the target platform.

In an increasingly connected world where more and more online devices are integrated into our everyday lives, a compromised device can potentially cause loss of user privacy, denial of essential services, and financial loss. For example, a security breach on a home gateway can redirect user traffic/data to a malicious server and compromise users' personal data. A well implemented secure boot process, as the main pillar of a secure and trusted environment, can harden the platform and block local and remote attacks caused by malicious code installation.

Building on top of the secure boot process, a trusted boot process can be deployed where the authorization to run (i.e., that only the processes with well-known measurements are allowed to be run) is also checked for all of the components, including the boot loader, kernel, images, drivers, master boot record, and all files that are referenced or executed during boot. The principle of trusted boot is "trust but verify." In other words, the goal of trusted boot is to offer evidence that shows the system behaves as expected.

A trusted boot process can be achieved by using system measurements with security modules such as the Trusted Platform Module (TPM), where code hashes, data value, and system configuration are sealed and reported by the root of trust (RoT). Because a TPM has controlled access even from the platform on which it is installed, TPM functionalities and storage are protected against possible abuses even from the integrating host. Specifically, a trusted boot process combines the use of a chain of trust (CoT), measurements for each boot component, and hardware protections to deliver local or remote attestation logs that can be checked against specific verifiers' policies.

Ultimately, unlike secure boot, the trusted boot device is measured and attested against expectations (prior measurements). In fact, whereas secure boot only performs integrity checking during the boot process and prevents modified software from being executed, the trusted boot process requires the use of local attestation (e.g., loading an operating system driver component) or remote attestation (i.e., checking the integrity measurements against the set of authorized values) to verify the process' authorization to run on the host (or for a device to communicate over the network). Moreover, when external (to the device) verifiers are used, remote attestation must be supported in order to be able to extract the measurements and securely deliver them to the verifier.

1 Secure Boot Architecture

1.1 Principles of Secure Boot Security

As mentioned earlier, the aim of a secure boot process is to guarantee that the software being executed on a device was checked for integrity and authenticity before execution began. In many broadband specifications, the process of retrieving and validating the firmware (also referred to as CM Code File) is known as secure software download (SSD).

In a modern embedded platform, there are usually two flash banks for storing firmware images. The active bank contains the image running on the platform, and the inactive bank contains a backup firmware image. If, for any reason, the image in the active bank becomes corrupted or not bootable, the device switches to the inactive bank (making the active bank inactive and the inactive bank active) and boots from the inactive bank. When a device is upgraded to a new firmware image, the new image is installed in the inactive bank.

In broadband's SSD process, the firmware is first validated against manipulation and authenticity by verifying the signature and the associated Code Validation Certificate (or CVC) chain before enabling the SSD process. After positive validation and successful installation of the new firmware in the inactive bank, the device switches the bank and boots from the newly installed firmware image that is signed for secure boot (SB). See Figure 1 for a typical SB and SSD code signing process. Note that the SSD signature is typically different from the SB signature(s). Both signature validations will block unauthorized attempts to modify the code after it has been signed, but the SSD signature is verified only once during the firmware upgrade, whereas the SB signature is verified at every reboot.

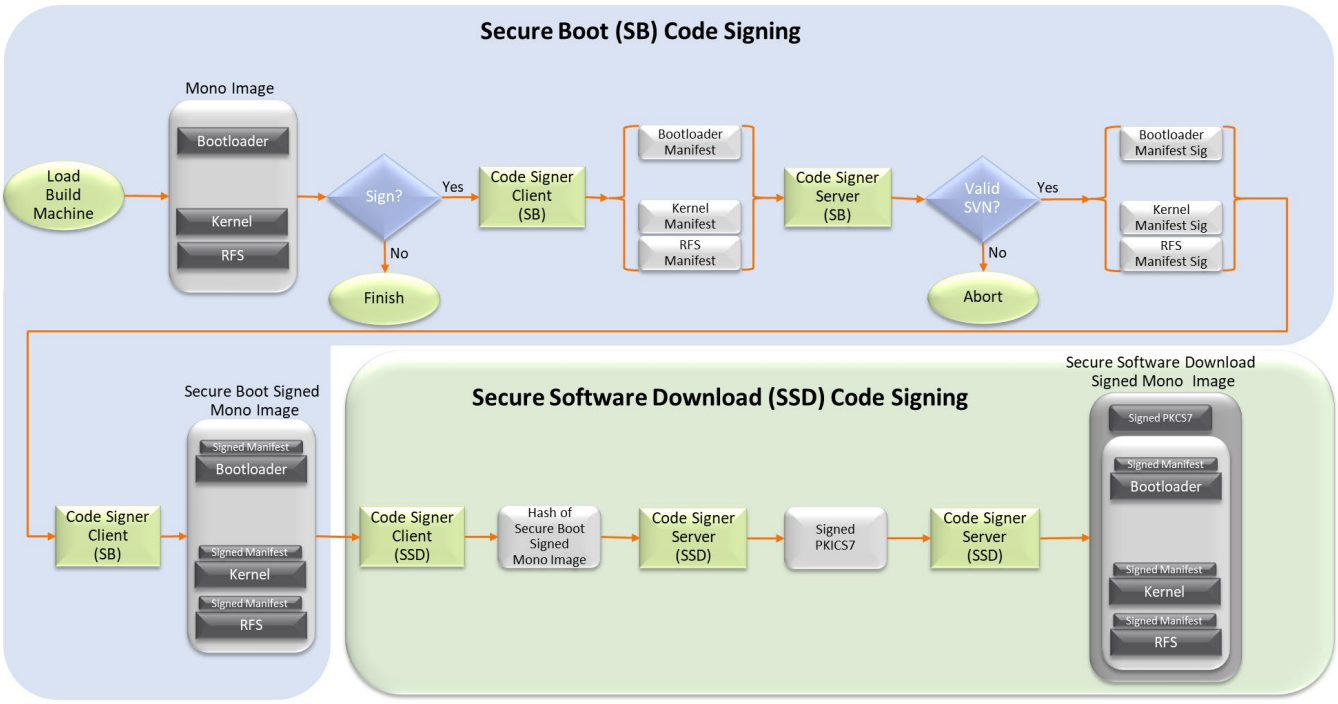

Figure 1: Typical SB and SSD Code Signing

1.1.1 Multi-Stage Secure Boot

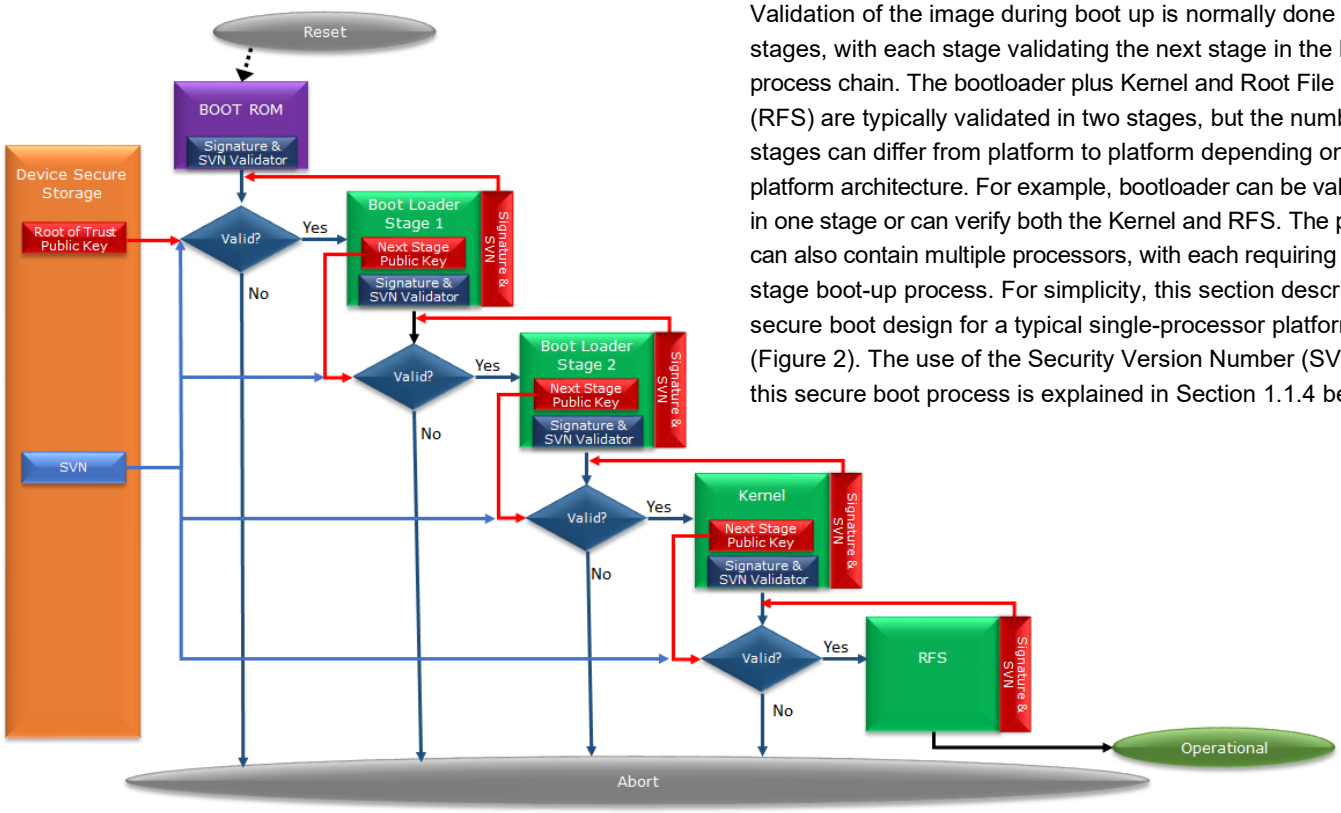

Figure 2: Typical Multi-Stage Secure Boot Flow for Embedded Devices

Validation of the image during boot up is normally done in stages, with each stage validating the next stage in the boot process chain. The bootloader plus Kernel and Root File System (RFS) are typically validated in two stages, but the number of stages can differ from platform to platform depending on the platform architecture. For example, bootloader can be validated in one stage or can verify both the Kernel and RFS. The platform can also contain multiple processors, with each requiring a multi-stage boot-up process. For simplicity, this section describes a secure boot design for a typical single-processor platform (Figure 2). The use of the Security Version Number (SVN) in this secure boot process is explained in Section 1.1.4 below.

To be able to validate the integrity and authenticity of boot images, a root of trust (RoT) must be securely programmed into the device at manufacture. The storage requirement for the RoT will be described in Section 1.2 The RoT can be a public key or a hash of a public key, as described in Section 1.2.

Once the RoT is established on the device, secure boot is enabled on the production line. On the subsequent boot up, the device looks for the RoT to validate the first stage of the boot up. If it does not find a valid RoT or if the first stage is not properly signed, the platform will abort booting up from that image. If the first stage is verified successfully, then the first stage will try to validate the second stage. The public key to validate the signature is part of the signed stage 1 image, which is already validated by Boot ROM. If the second stage is verified successfully, then the second stage will try to validate the third stage. This process continues until all stages are verified and the device becomes operational. If the signature validation fails at any stage of the boot up, the device aborts the boot up and tries to switch the flash bank and boot from the other flash bank.

1.1.2 Multi-Key Code Signing Configuration

Each stage can have its own signing key, or multiple stages can share the same key, e.g., stage 1 shares its signature verification public key with stage 2, but stages 3 and 4 each have unique verification public keys. The signing key(s) can also be customized for each customer/service provider. In the multiple-key approach, the code signers can be separated per image component, per customer, or per a combination of the two, so the risk of system compromise would be reduced if a component is compromised. The code signing policy of an organization should define the number of keys.

1.1.3 Code Execution from DRAM

One of the principles of secure boot is to execute verified code from DRAM, which requires the platform to install and verify the entire image in the DRAM at boot up and not go back to the flash memory (non-volatile memory, NVM) to retrieve code at runtime. Use of unverified code from NVM at runtime is forbidden. Using DRAM, however, requires the platform to allocate enough space in the DRAM to accommodate the entire image. When the platform does not have enough DRAM to load the entire image, it has to swap in and out a portion of the image in the DRAM. One prime example is when the size of the RFS is too big to fit into the DRAM. In this case, the platform would have to swap in and out portions of the RFS at runtime, so the source of the RFS would be NVM at runtime. A common solution to this problem is to divide the RFS into blocks, generate a hash for each block, and create a hash tree where the root of the hash tree is stored in the Kernel while the entire image is built. When the Kernel swaps in and out blocks of RFS, it will also validate the hash of a block from the node all the way to the trusted root of the hash tree. Linux DM-Verity provides this authentication mechanism.[1]

1.1.4 Anti-Rollback for Firmware

Updated firmware images are released as part of a product's life cycle, but the updates present a chance for unintentionally introducing a sevrer vulnerability that escapes quality control testing into the firmware. Such an event can potentially result in a secure boot signed image with a vulnerability that can compromise a mass population of devices in the field—the signed firmware image with the vulnerability can be installed on the device even after the device is upgraded to a new firmware image with the fix. The assumption is that if the signed firmware image with the vulnerability is released, it will be posted on the Internet and will be available to bad actors. Even after a new signed image with the fix is released and installed on a device, a bad actor can force the device to upgrade to the vulnerable firmware image. Because the vulnerable firmware image is signed properly, the device does not complain and boots up from that firmware image.

This issue can be mitigated by what is called anti-rollback. Each secure boot signed firmware image contains a security version number (SVN), which is part of the signed image and is verified by the device during the secure boot image validation at boot up. The device also keeps the current value of the SVN in its secure storage. The SVN is incremented only when there is a severe vulnerability in the firmware, and it would require approval and close coordination of the equipment vendor and the operator/service provider. Otherwise, an uncoordinated SVN update can cause severe field issues. An SVN may or may not be updated during the life of a product. If it is updated, it would be seldom and for only the most severe vulnerabilities. If a critical vulnerability is fixed in a new release of the firmware, the SVN is incremented in the new signed firmware image. The device always checks the firmware SVN against the SVN value in its secure storage during the boot up as part of its secure boot process, so once this firmware with the incremented SVN value

[1] Linux kernel user's and administrator's guide: dm-verity

is installed on a device, the device will fail to boot up from any previously signed images with a smaller SVN value. The SVN validation logic typically works as follows.

1. If the firmware SVN is less than the stored SVN, the device rejects the firmware.
2. If the firmware SVN is equal to the stored SVN, the SVN is validated successfully.
3. If the firmware SVN is greater than the stored SVN, the stored SVN is updated with the firmware SVN and the SVN is validated successfully.

The SVN validation logic prevents any older firmware image with a vulnerability from booting on a device ever again.

1.2 Physical Security of Keys in the Device

Cable modems, residential gateways, access points, and Internet of Things (IoT) devices often use a monolithic image containing all the software necessary for the system to boot and run. Such devices might be referred to as embedded or monolithic systems. A typical secure boot design for these systems often utilizes asymmetric public key signatures at each boot stage and has the advantage of not requiring confidentiality protection for any secret values. Encryption may be optionally utilized to protect the intellectual property of software releases and confidential user data, but it is not needed for the purposes of secure boot because signature validation makes use of only public keys.

Secure boot depends on the integrity of the code verification RoT stored in each device. Typically, for example, as defined in the UEFI secure boot specification, it will be a root certificate authority (CA) certificate or a root public key that is utilized to authenticate the First Stage Boot (FSB). This RoT is either self-signed (if it is a certificate) or not signed at all. In both cases, the integrity depends on the security of the device's non-volatile storage (e.g., flash or on-chip static RAM) and how it handles unauthorized attempts to tamper with the device and replace this RoT.

Secure storage with integrity protection may be applied either directly to the RoT value or to its hash value. For example, an SHA-256 hash of the RoT value may be stored in integrity-protected memory while the actual RoT (certificate or public key) may be stored anywhere, including the header or footer associated with the FSB image. The RoT has to be revalidated during each boot cycle by comparing its hash value against the hash value in secure storage.

Typical methods to provide integrity for a secure boot RoT (public key or its hash) include the following:

- burning an RoT value into one-time programmable (OTP) fuses or eFuses, which cannot be altered after they are programmed, typically done in the factory;
- including an RoT value in ROM;
- storing an RoT value in a protected sector of secure flash;
- including an RoT value as part of a separate security processor's code image or internal on-chip secure memory that cannot be altered by software running outside of that security processor; and
- storing an RoT value in standard, non-secure, non-volatile memory that is encrypted or with a Message Authentication Code (MAC) such that decryption or MAC verification can be performed by a specialized security processor prior to the start of a secure boot cycle.

The above list is far from exhaustive; secure boot and integrity-protected storage techniques are highly variable and depend on the chip provider. Each approach has implementation and security pros and cons and should be thoroughly understood by developers. Personal computers and general Linux and Windows server platforms may include UEFI secure boot or a chip-vendor-specific secure boot solution. Alternatively, vendor-specific solutions may be used.

1.3 Root of Trust Revocation

Having a single immutable RoT in a device may be too inflexible. If the private key corresponding to an RoT is ever compromised, secure boot is also fully compromised. This risk can only be mitigated by preventing the device from using the compromised key for validations and enabling the use of an alternative RoT taken from the ones built in at manufacture. It is important in such a case that a

device prevents an unauthorized switch to the wrong RoT, such as one that has already been revoked or that corresponds to a different customer. Some of the protection mechanisms utilized today to correctly handle multiple roots of trust include Reply Protection Memory Blocks (RPMBs) or Secure Execution Environments.

Multiple roots of trust are programmed into read-only non-volatile memory; additional OTP fuses point to the currently valid/enabled RoT. For example:

- 0b0000 (when none of the fuses are programmed) = RoT #0 is enabled
- 0b0001 (after one of the fuses is programmed to a 1) = RoT #1 is enabled
- 0b0011 (after a second fuse is programmed to a 1) = RoT #2 is enabled

With this technique, each fuse can be programmed from 0 to 1 and cannot be changed afterward. Once RoT #1 is enabled, RoT #0 is revoked and will never be used again. Once RoT #2 is enabled, RoT #1 will never be used again. Additional roots of trust can be enabled and revoked in the same way.

When the RoT value is stored in a secure sector of flash protected with an RPMB feature, the RPMB will allow secure authorized updates to this secured area of flash with anti-replay protection. Flash with an RPMB feature will prevent an old version of flash content to be reloaded. Updates to a protected area of flash would generally be performed inside a Trusted Execution Environment (TEE) on the main processor or separate security co-processor, and some of the details may be proprietary to the chip vendor. RPMB specifications for eMMC flash are available from JEDEC.[2]

When the RoT is included in a security processor code image or in non-volatile memory that is validated by a security processor, each chip vendor may have their own proprietary method to prevent anti-rollback to an older RoT that may have been revoked. It generally involves versioning the RoT and/or security processor code image and rejecting older versions of the RoT that may have been previously revoked.

In addition to anti-rollback protection for a secure boot RoT, anti-rollback protection is needed at every boot stage. Once a serious software security vulnerability is found and subsequently fixed, a security mechanism is required to prevent adversaries from reloading that older, insecure version of the software. See Section 1.1.4 on anti-rollback techniques. Software validating the next boot stage needs to make sure that its version number is not smaller than the minimum version number allowed for that boot stage.

The above techniques for preventing rollback attacks on an RoT can also be applied as anti-rollback techniques for each boot stage. For the most flexibility, each boot stage should have its own version number that is compared to a separate integrity-protected minimum version number.

Alternatively, in a simplified design, a single minimum version number could be maintained for all or some of the boot stages combined together. In that case, when one of the boot stages requires a serious security vulnerability fix and the minimum version number is incremented, all of the boot stages have to be released together with a new version number.

1.4 Code Signing Procedures

At a high level, authenticated software can be considered as going through multiple stages in the software lifecycle, including software development, software build, code signing, installation, execution, and (eventually) deprecation. Software development typically utilizes a source control system, allowing developers to design and implement software based on functional requirements. The software build system compiles and combines developed source code to generate a software package. In systems using secure code distribution, the resulting software package is signed, meaning that a digital signature is created from it by using a code signing key. The signed software package includes the original software package and the signature in some format. The final signed package is then deployed to the target execution environment, where the signature of the software package is verified before it is accepted for execution.

Because adversaries typically attack the weakest link in the system, all stages described must be adequately secured to prevent attackers from compromising the integrity of the software. In this section, we discuss the code signing process in more detail. Having a

[2] Universal Flash Storage, Version 3.1, January 2020

secure code signing process without also protecting the software build process or target execution environment will only provide a false sense of security. Code verification steps prior to signing a production code release should, at a minimum, include the following:

- performing code reviews and looking for programming mistakes that can cause security vulnerabilities, such as buffer overflows, not checking for array boundaries, or using unsafe APIs that do not check for memory errors;
- utilizing automated tools for checking source code that can identify the majority of such programming mistakes; and
- utilizing automated tools that validate running executables for known security vulnerabilities and can identify missing input validation.

Focusing on the code signing system or process, the primary function is to sign code, which can be achieved in many ways with different levels of complexity. On one extreme, the code signing key can be stored as a password-protected file on a development machine, and code signing occurs when the developer executes a signing command. Regardless of how it is implemented, every code signing system or process should include an audit trail that identifies the who, when, and what of all the signing activities; protect the signing keys against disclosure or compromise; and should allow system administrator to control and manage security parameters. For instance, an audit trail would allow a malicious signed image to be traced back to the responsible party.

A code signing system would typically log only the hash of the binary code submitted for signing to reduce the associated storage overhead. It may furthermore be operated by a third party that is not privileged to access the software provider's source code and even binary code. Development and build systems on the other hand include source code files used in a build of a software package, version information, and which source code updates were made at which time by which user. Such information should be integrity-protected and is just as important as the code signing system logs for a potential investigation as to how a security vulnerability was introduced into a software release.

To identify the party responsible for submitting code for signing, the code signing process or system should have a way to identify the user performing each signing activity. For example, the system may require a user to be registered and logged in before they can submit a code image for signing. In addition to authentication, proper authorization should also be enforced as different users are typically responsible for different projects. The principle of least privilege should apply—a user should be granted permission to sign using a particular signing key only if they is responsible for software development for the project or product associated with that key. Even for the same product, for example, a developer responsible for the boot code may not be responsible for the platform or application code.

The code signing system may need to be automated; for example, code could be built automatically on a regular basis as developers continuously update different modules. Automated code builds carry additional security risks and will not provide the same level of detail in an audit trail. Nevertheless, it may not be possible to avoid them, so it is important in this case to identify the build machine as the code signing client and tie the signing activities to that machine. Developers are logged in to the machine to submit their code, and those activities should be logged in the build machine accordingly. Together, the logs from both the build machine and the code signing system will provide a full picture of who contributed to what code image to be signed.

It is of utmost importance to keep code signing keys confidential. If a code signing key is leaked to an attacker, they can use it to sign any malicious code and therefore defect the purpose of code signing and secure boot altogether. Obviously, storing code signing keys as data files on a computer where many developers have access provides very little control over the signing keys. The keys are susceptible to exposure and cloning. There are many examples of known signing key compromises, including ASUS and Bit9.[3] The best practice to restrict access to the code signing keys is to protect them by using a hardware security module (HSM). A code signing system can then control access to the hardware-protected signing keys to only authenticated and authorized users. It can also make sure that the input code is of a specific format and not arbitrary data.

In addition to signing keys, the system can also control some security-related parameters. It is standard practice to have a security version number associated with a code image. If a vulnerability is discovered, it is possible to sign a new version of the code with the vulnerability fixed (see Section 1.3 for more details). The code signing system can control such parameters by either verifying that the security version number in the code is as expected or inserting a configured security version number into the input code prior to performing a signing operation. Other examples of security-related parameters that can be controlled by the code signing system include a product or model identifier, market segment identifier, and organization identifier.

[3] ASUS: "Code Signing Compromise Installs Backdoors on Thousands of ASUS Computers," April 2019, The SSL Store—Patrick Nohe
Bit9: "Security Firm Bit9 Hacked, Used to Spread Malware," February 2013, Krebs on Security
The SANS Institute describes additional compromises: "The Scary and Terrible Code Signing Problem You Don't Know You Have," November 2014, Sandra Dunn

This section provided a brief overview of the functions that a code signing system or process should have. Note, however, that the full code signing infrastructure also needs to have adequate physical, network, and system security in place. All network devices and physical hosts should be hardened according to the latest security guidelines in the industry, and regular and extensive network scanning and penetration testing should be in place to minimize any remaining security vulnerabilities available to an attacker. Though the investigation in the recent SolarWinds SUNBURST vulnerability continues, that cyber security event illustrates the importance of securing your code distribution environment. A summary by the Center for Internet Security provides details that show the impact of this vulnerability.[4] A summary of the attack time line and the resulting devastation is also available.[5]

1.5 Maintaining Security During Software Development

It is a common practice during a software development process to execute and test code that is not yet fully tested or qualified for a production release. A code image may have incomplete or missing features or incorrect/modified logic that, if exposed, may be exploitable by attackers. Therefore, it is important that only production-grade code is signed with a production signing key. For test code, one solution is to use a test chain of trust for the code signing keys. Dedicated hardware may be configured specifically with a test root of trust as opposed to the production root of trust. In this case, all development code images are signed using code signing keys chained to the test root of trust. As a result, a signed test image will be usable only on test hardware, not the production platform. Alternatively, on the test platform, secure boot code could be configurable with the help of a signed configuration message to temporarily allow test-signed or unsigned code. By maintaining the cryptographical separation between test and production signing keys, one can avoid polluting the production environment with potential vulnerabilities that are present in test code.

1.6 DOCSIS Cable Modem Secure Boot Example

The DOCSIS 4.0 security specification (SECv4.0) now includes secure boot requirements, which cover integrity enforcement for all software layers and physical security requirements for the secure boot RoT. The DOCSIS 3.1 specification currently does not require secure boot, but some cable operators with greater focus on CPE security have already been requiring it for DOCSIS 3.1 networks, and there are DOCSIS 3.1 cable modems available today that meet the secure boot requirements of SECv4.0. Such cable modems incorporate a secure DOCSIS 3.1-capable system on a chip (SoC) that has the additional hardware security elements required for secure boot. Section 1.1, “Principles of Secure Boot,” provides a detailed example of a typical secure boot implementation that is applicable to cable modems.

1.7 Linux UEFI Secure Boot Example

UEFI (Unified Extensible Firmware Interface) secure boot[6] is a commonly available feature that can be turned on in a Linux or Windows PC BIOS. Configuration of a PC BIOS can be locked with a password, preventing an unauthorized user from turning off this feature. This type of secure boot is generally not applicable to embedded devices such as cable modems with specialized SoCs.

UEFI secure boot incorporates a code verification certificate hierarchy rooted at a platform key and maintained in a key exchange database (Figure 3, following page). This database provides distribution of verification signatures to three systems: an allowed signatures database, a revoked signatures database, and a timestamping database. The allowed signatures database is used to authenticate the “shim” and the Linux kernel.

The Platform Key (PK) is an RSA public key that is the root of the UEFI secure boot; it is incorporated into secure BIOS. The PK is used to verify a PKCS#7 signature on the Key Exchange Key (KEK), which in turn is used to validate PKCS#7 signatures on the signature database (DB), the Secure Boot Forbidden Signature Database (DBX), and the Secure Boot Timestamp Signature Database (DBT). These are all digitally signed public keys and are effectively digital certificates, although not in the X.509 format.

The DB is a list of trusted Code Verification Certificates (CVCs) and/or hashes of specific Linux or Windows bootloaders. An entry with an SHA-256 hash can verify a specifically approved bootloader release. An X.509 certificate listed in the DB may be either a CVC that directly validates a signature on a bootloader or any CA certificate in the CVC certificate chain.

[4] “The SolarWinds Cyber-Attack: What You Need to Know,” March 2021, Center for Internet Security
[5] “SolarWinds Orion Security Breach: Cyberattack Timeline and Hacking Incident Details,” June 2021, Channel e2e—Joe Panettieri
[6] Unified Extensible Firmware Interface (UEFI) Specification, Version 2.9, March 2021

The DBX is a list of revoked bootloader hashes and/or certificates (for any certificate in a CVC certificate chain). A bootloader with a hash or certificate found in the DBX is considered to be revoked and must be rejected. A new DBX then has to be signed and distributed to each applicable PC in order to enforce the revocation of the newly revoked bootloader releases or certificates.[7]

The DBT is a list of certificates of accepted timestamping authorities that can sign timestamps attached to a code signature.

Secure BIOS depicted in Figure 3 can be used to verify a PE/COFF (also known as Microsoft Authenticode) signature on a bootloader, which is a shim in this example of a Linux secure boot. The shim can use the same signature format and DB to verify a signature on the Linux kernel.

The Linux kernel in turn needs to verify the application code. It is generally not possible to validate all of the Linux application code on a general-purpose computer. However, a read-only file system such as rootfs may be validated with a Linux facility such as dm-verity.[8] When dm-verity is enabled, it will validate a hash of each page that has been loaded into physical memory for execution.

Thus, it is possible to configure even a general-purpose Linux computer with secure boot such that at least some applications that reside in a read-only file system are protected from unauthorized modifications.

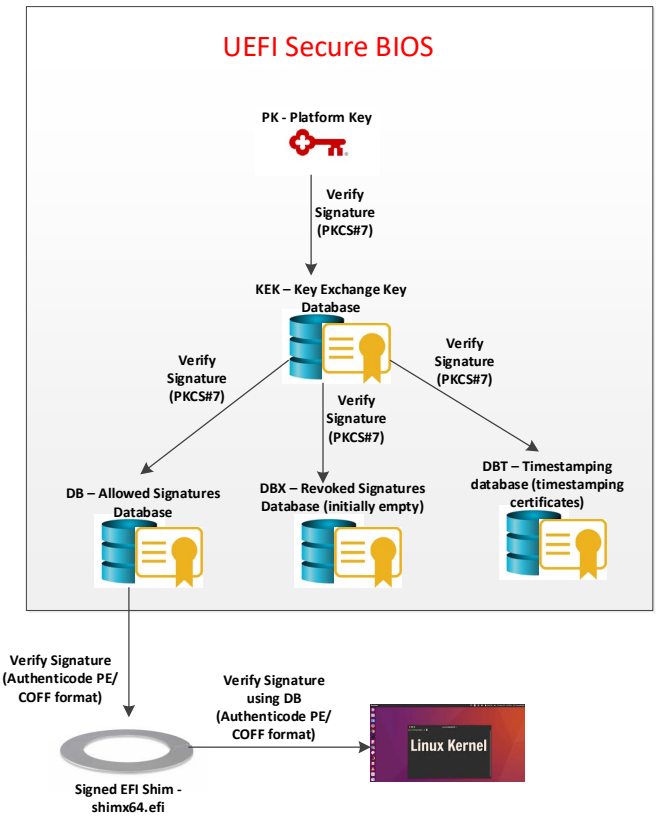

Figure 3: Linux UEFI Secure Boot Example

1.8 Agile Code Signing System

A code signing system should be designed to be algorithm- and format-agnostic in order to support code signing algorithms and formats required by various products, applications, and operating systems. It should allow new code signing algorithms and/or formats to be added without making significant changes to the underlying infrastructure. In general, a code signing system takes in a piece of code or code hash and returns a signed code image or a signature. Other inputs into the operation include signing keys (and encryption keys in some cases) and other security parameters.

The system design should be based on this high-level notion rather than specific algorithms or formats. This approach is particularly important as new post-quantum crypto algorithms are on the horizon. Current candidates for post-quantum algorithms typically have much larger public key and signature sizes when compared to traditional digital signature algorithms, which can create issues, especially for older environments. To be ready to switch to the new cryptography when and if needed, developers and system integrators should start looking today at how to accommodate the bigger sizes of these data objects for the different parts of the system (database entries, authentication protocols' fields, etc.).

1.9 Software-Based Secure Boot

The root of trust for secure boot is generally protected in hardware in a secure SoC, secure microcontroller, ROM, secure flash, or some other secure hardware element. Sometimes, however, a product does not have any of those hardware security capabilities; they may have been omitted because of cost considerations, or an application may be running in a cloud environment on ubiquitous commercial off-the-shelf (COTS) hardware.

[7] DBX entries are supported both commercially and in the open-source world. For example, Linux distributions, such as Ubuntu, use DBX entries to prevent the boot of buggy versions of Grub (see https://wiki.ubuntu.com/SecurityTeam/KnowledgeBase/GRUB2SecureBootBypass#DBX_Update). The official list of DBX entries can be downloaded from the UEFI website (https://uefi.org/revocationlistfile).

[8] Linux kernel user's and administrator's guide: dm-verity

Simply adding a digital signature to software in that environment has very limited security value. Even when every software layer is digitally signed, there is still a question as to how to protect the logic and the public key value needed to verify the signature on the first boot stage. They may be hidden from regular system users, and OS-level access may be restricted, but operating systems and application software are generally riddled with security bugs, including the ones that enable an unauthorized user with root access. Software vendors regularly issue security patches, but an adversary can take advantage of a security bug before a security patch can be installed or before that security patch becomes available. Without additional protection, it is just a matter of time until a root of trust is replaced or secure boot is completely bypassed or disabled.

Fortunately, several software security tools are available that can provide improved security.

- Software obfuscation—It can be applied to the lowest level boot stage to hide the logic and public key utilized to verify the signature of the next boot stage. Remaining boot stages are digitally signed, and any offline tampering will be detected during a signature check. They may still benefit from obfuscation to make runtime tampering more difficult.
- Integrity validation—Though no hardware element can verify the signature of the lowest level boot stage, there are security tools capable of inserting digital signatures that cover portions of the software at random locations. Software will then utilize a security library to verify such signatures as they are encountered. This type of integrity checking is made to be difficult to find and remove, thus providing tamper resistance. Effectiveness of this technique varies widely with the different tools, but it only needs to last until the next software release. Each software release can change where and how those signatures are inserted, so an adversary would have to repeat the process in trying to remove them. A version or a timestamp check or just an addition of critical or desirable software features can be utilized to prevent the use of an older software version that had its integrity checks bypassed.
- Anti-debug—Software security tools can detect the presence of a debugger and either halt execution or intentionally alter normal behavior while running in the debugger to confuse the adversary. Such tools will make attempts at reverse engineering and removal of software integrity checking more difficult.

The following papers provide more information on such software security tools.

- Dynamically Addressing the Gap of Software Application Protection Without Hardware Security
  R. Shamsaasef, A. Anderson, SCTE-ISBE Cable-Tec Expo 2019, September 30–October 3, New Orleans, LA
- Cloud-Based Dynamic Executable Verification
  R. Shamsaasef, A. Anderson, S. Medvinsky, SCTE-ISBE Cable-Tec Expo 2020, October 12–15, Virtual Experience

2 Trusted Boot Architecture

2.1 Principles of Trusted Boot Security

In cybersecurity, proactive strategies prevent major incidents before they happen, which requires insight and experience in identifying and addressing security risks. On the other hand, the reactive security approach mitigates compromises that are in progress. Both approaches are used to provide comprehensive defense mechanisms. The goal of this section is to offer proactive and reactive security principles for trusted boot deployment.

In this section, the proactive security principles include the following.

- Practice minimalism for the root of trust (RoT).
- Use a stand-alone hardware RoT.
- Perform Trusted Platform Module (TPM) pre-attestation for installed keys and certificates.
- Practice defense in depth.
- Use fresh and comprehensive measurement for attestation.

The reactive security principles include the following.

- Use an appropriate certificate and key revocation mechanism.
- Enable local attestation when needed at boot-time.
- Always use remote attestation after boot-time.
- Use constrained policies for disclosure.

2.1.1 Minimalism for Root of Trust

The RoT comprises the embedded trusted primitives that provide the basis of other security controls. These primitives must be trusted because they cannot be measured—they are treated as fact. They are the foundation of the chain of trust (CoT) that is created during the boot process. Usually, the RoT is required to offer the following minimum and necessary security functionalities:

1. protection of cryptographic keys,
2. an initial measurement and validation process, and
3. operations that are linked to protected keys and system measurements, which should be isolated and logged by a certain device, such as a TPM.

To provide these security functions, there are three types of RoT.

1. The Root of Trust for Storage (RTS) provides a protected storage area for keys and sensitive data.
2. The Root of Trust for Measurement (RTM) performs the initial measurement process.
3. The Root-of-Trust for Reporting (RTR) reports the system measurement to the bonded TPM that is used for attestation.

In modern architecture, it is common to have software components and firmware come from multiple suppliers. The manufacturer of the platform places its trust authority within a Core-RTM (CRTM), where the initial measurement happens and where the code remains unchangeable. As software execution transitions to the code outside of the CRTM, the CoT becomes the tool of a boot process to build the hierarchy of trusted software layers (Figure 4). The CoT is established during the trusted boot process to transfer data and controls between each boot stage by measuring and recording its measurement into an event log at first. The integrity of each measurement result is then sealed with the cryptographic keys protected by the RTS, and the result is reported to a remote verifier by the RTR.

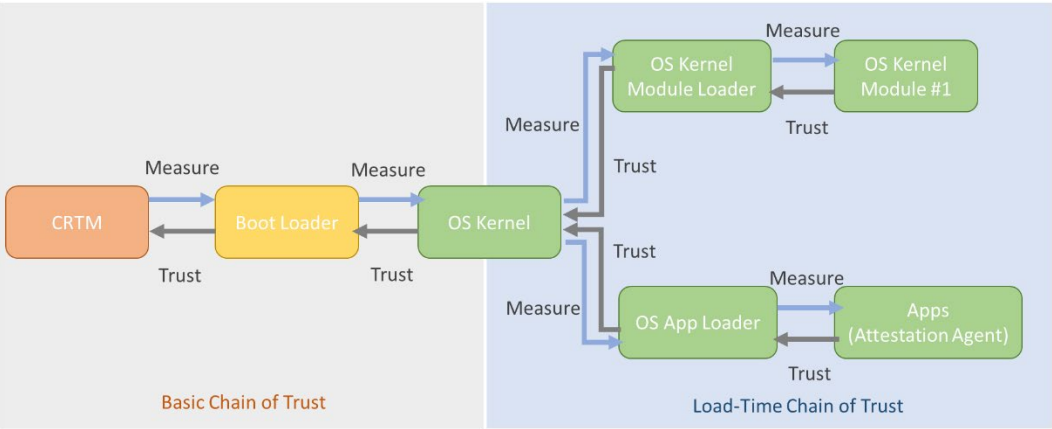

Figure 4: Chain of Trust

Because the RoT is the trust anchor of the CoT, it is important for the RoT to remain immutable. Otherwise, the whole CoT may be broken and impact the integrity of the booting process. Without the integrity of the RoT, it is meaningless to continue building the trust hierarchy with the CoT.

2.1.2 Stand-Alone Hardware Root of Trust

2.1.2.1 Boot Firmware Rootkit Attack

As the first code that executes at power-on, boot firmware is in fact a root of trust in modern systems. One threat encountered when using this software component as an RoT is the rootkit attack that injects malicious code at a lower level before the anti-malware for detection, such as ZeroAccess,[9] is executed. Malicious code can be injected at the lowest level of the software stack; a paper presented at the PacSec 2013 conference[10] gives examples of BIOS infecting malware and the available mitigations.

A possible method for remediation is to generate the hash value of the firmware and sign it with the manufacturer's key, requiring the booting process to always verify each boot stage before transitioning execution to the OS and application software. Using a hardware-based root of trust (HBRT) as a stand-alone module to the device can help prevent such an attack because its identity is isolated from the device memory and the authentication credential is not writeable by the main CPU. Additionally, an HBRT uses physical tamper resistance techniques like encapsulation and data bus scrambling to prevent the exposure of the Low Pin Count (LPC) bus from sniffing and decoding.

2.1.2.2 Random Number Attack

An RoT has an obligation to also provide minimal, necessary functionalities such as cryptographic key generation and management. Many algorithms such as RSA require random numbers to generate keys. If an attacker knows the pattern of random seed generation and the algorithm for key generation, it is possible they can generate the same key without breaking the key protection in such a device. One example of such an attack is mining for RSA public keys that share a common prime factor because a poor-quality random number generator is in use. With a common prime factor, the rest of the divisors of those keys can be calculated. Researchers used the greatest common divisor (GCD) function to find that distinct RSA moduli N1 and N2 have common prime factors. Because GCD can be computed in milliseconds, an attacker can easily recover the corresponding private keys. This is a serious vulnerability affecting IOT devices—research found that 1 out of every 172 sampled RSA certificates shared a common factor with another.[11]

Unfortunately, with only software, most computers can only generate a pseudorandom number with the system time and deterministic algorithms. A passive eavesdropping attacker can perform a randomness test and find the pattern through a list of randomly generated numbers. With an HBRT, random numbers can be generated through a True Random Number Generator (TRNG) that uses a physical process such as thermal or signal noises as the seeds. Additionally, an HBRT includes a secure clock or secure counter for time measuring, which avoids attacks that violate time-based policies.

2.1.2.3 Trusted Platform Module as HBRT

Trusted computing technology proposed by the Trusted Computing Group (TCG) aims to build a trustworthy environment so that improper software and hardware authentication and authorization can be verified and detected. As a result, the TCG has introduced TPM (Figure 5) and related software to be the trusted root of the system throughout the boot process. Details are presented in the TPM 2.0 specifications.

TPM chips are not yet widely deployed in many devices. Not all IoT systems and components adopt a TPM or an equivalent secure hardware module. To address this, the TCG created the Device Identifier Composition Engine (DICE) working group to enhance security and privacy with minimal silicon requirements. In addition, Global Platform defines the Root of Trust Definitions and Requirements as well as the IoTOPIA framework for IoT devices. PSA Certified is an alternative security framework for ARM-based IoT devices.

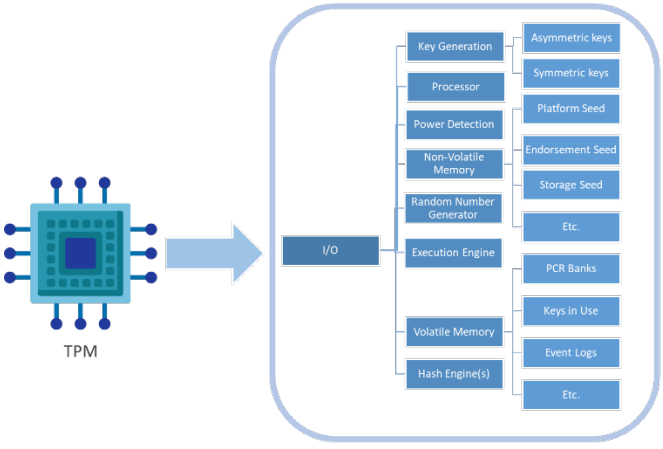

Figure 5: TPM

[9] "ZeroAccess Indepth," October 2013, Symantec–Security Response—A. Neville, R. Gibb
[10] "Defeating Signed BIOS Enforcement," C. Kallenberg, J. Butterworth, X. Kovah, S. Cornwell, PacSec 2013, November 13, Tokyo
[11] "Securing RSA Keys & Certificates for IoT Devices," December 2020, Keyfactor—J.D. Kilgallin

2.1.3 Key and Certificate Protection

2.1.3.1 Defense in Depth

The defense in depth approach uses layered defensive mechanisms to protect systems or data. With layering, if one defense fails, another is there to block an attack. Defense in depth is used in trusted boot where the whole booting process is considered to be a sequence and different stages use corresponding keys for signing and encryption.

In trusted boot, the RTS shields TPM memory from unauthorized access and secures storage for the TPM owner. To shield access, the user needs to "take ownership" of this TPM. During the initialization of the TPM, a user may optionally first set a TPM password so that, in the future, the same user can use this password to prove their ownership and access the memory. Once the ownership is set, no other user can access, disable, enable, or clean this TPM without first showing their ownership. This mechanism is used to create the Storage Root Key (SRK). This TPM then uses the SRK to generate storage keys and binding keys for encrypting data, as well as attestation identity keys (AIK) for remote attestation. An SRK must be stored in non-volatile memory as the root of a key hierarchy and never leave the TPM. An SRK is unique and bonded with a specific user and that user's TPM, so the only way to generate a new SRK is to clear this TPM and reset the ownership.

Alternatively, to avoid manual password entry, the default TPM password may be left as NULL. Once encryption keys are created within a TPM chip and are utilized to encrypt critical resources in storage, the key values cannot be read out. Secure boot in this case is expected to prevent injection of unauthorized software that may attempt to decrypt and expose those critical resources. In other words, the function of a TPM password on a general-purpose PC platform is replaced by a secure boot process on an embedded device.

The Endorsement Key (EK) is the RSA key pair used to prove the RTR's identity to a genuine TPM. The Endorsement Primary Seed (EPS) is used to generate EKs and a specific known EPS value can only be only injected by TPM vendors. Platform authorization is required to change the EPS to a newly generated random value. After the EPS has been modified, all objects in the EK hierarchy are invalided. The EK private key is used to decrypt values that are encrypted by a public EK key. The EK private key is stored inside the RTS and is never visible outside of the TPM.

TPM keys may be applied to different defense mechanisms, including measured and trusted boot, device attestation, and secure key storage. TPM-protected cryptographic keys may be classified as migratable and non-migratable. Migratable keys are not bounded to a specific TPM and can be utilized outside a TPM or moved to another TPM, whereas non-migratable keys are unique, bonded with a specific TPM, and can never be exported to another. The migratable key chain is designed so that only one key, the Legacy Key (or the Platform Migratable Storage Key), needs to be migrated in order to take all the keys shown in Figure 6 into a new TPM.

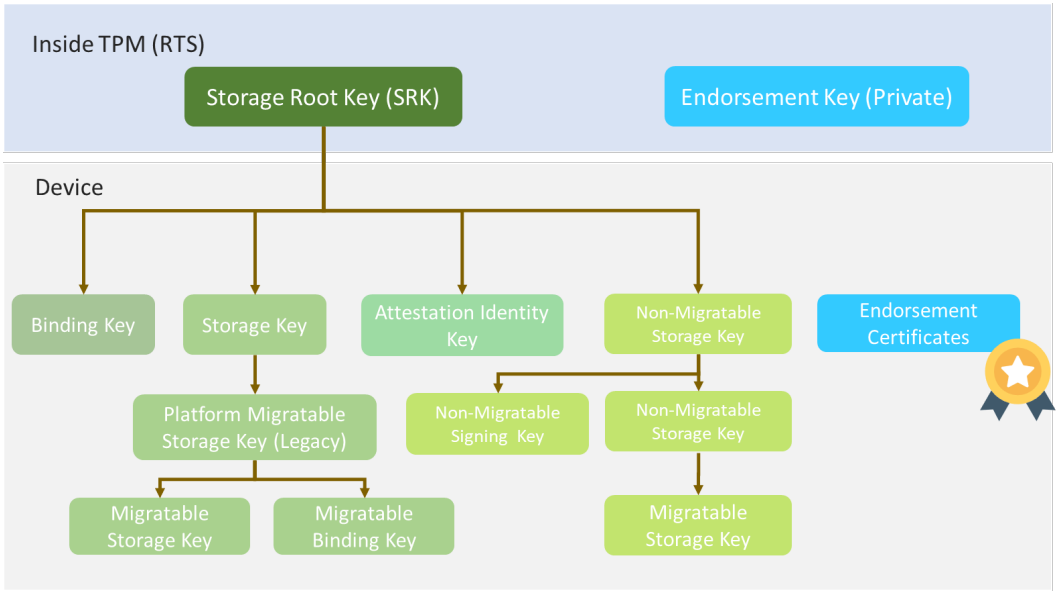

Figure 6: TPM Key Hierarchy with an RTS

2.1.3.2 TPM Pre-Attestation

It is important to ensure that the TPM is genuine and has a valid identity before starting to use it for measurement and attestation. This identity is the EK and EK certificate and should never leave the TPM itself for external use. Before shipping the TPM, the manufacturer needs to issue the EK certificate that validates the unique identities of the device and the device manufacturer. Alternatively, instead of shipping the TPM chip with an Endorsement Certificate that may be compromised and need to be revoked later, an Endorsement Seed can be embedded at the factory and used to recreate the Endorsement Certificate while in the possession of the customers, as long as they have the certificate of authenticity.

In the first few years of the TPM 1.2 standard, only several vendors chose to provide Endorsement Certificates, and a few did not even provide an EK for TPM-enabled platforms, which is critical for device authentication. It took years for many TPM manufacturers to start shipping devices with TPM 1.2 Endorsement Certificates. Therefore, for the new and much more complicated TPM 2.0 standard, TCG provides TPM 2.0 provisioning guidance for TPM manufacturers, platform manufacturers, and platform administrators.[12]

The purpose of the EK pair is to prove that the corresponding private EK is stored in a genuine TPM and acts as a unique identity. So, in keeping with the defense-in-depth policy to control the exposure of the private EK, the private EK must only be used to decrypt secrets. Consequently, it is not a signing key and therefore should not be used to sign any data.

Because the private EK must not be used for digital signatures, the attestation keys are used for signing a measurement that is sent to a remote verifier. The steps shown in Figure 7 should be performed when installing the TPM to the system as the key pre-attestation.

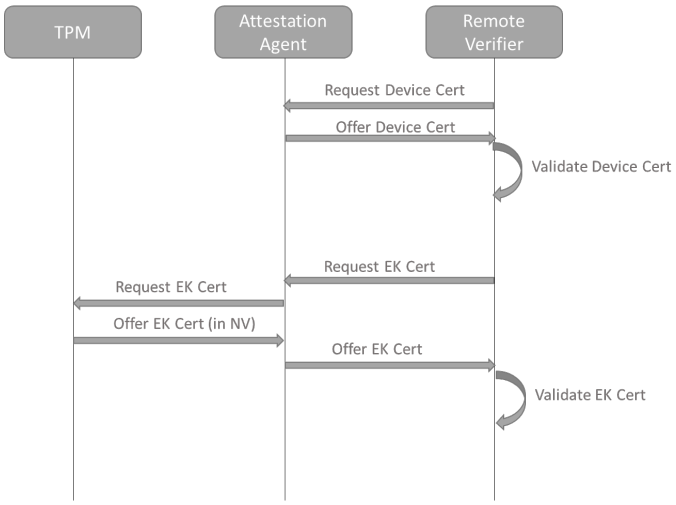

Figure 7: TPM Pre-Attestation: Link TPM with Device

2.1.3.3 Use Appropriate Certificate and Key Revocation Mechanism

Unfortunately, many researchers have demonstrated the possibility of recovering keys from a sealed TPM. One example is TPM fails,[13] in which researchers collected execution time samples based on elliptic curves during TPM signature generation. The researchers used these samples to filter the signature by leading zero bits and then used lattice construction algorithms to recover the private key. The attacker in this case would need to take control of the host software to submit multiple messages to the TPM chip for signing; with or without the key recovery, the corresponding certificate should be revoked. Another example is PS3 Epic Fail.[14] In this case, the researcher found the developers had used a hard-coded random value for generating the ECDSA signature, which gave the hacker the possibility of recreating the private key and breaking the signing procedure.

[12] "TCG TPM v2.0 Provisioning Guidance," March 2017, Trusted Computing Group
[13] "TPM-Fail: TPM Meets Timing and Lattice Attacks," November 2019, TPM-Fail—D. Mohimi, B. Sunar, T. Eisenbarth, N. Heninger
[14] "Hackers Describe PS3 Security as Epic Fail, Gain Unrestricted Access," December 2010, Exophase—Mike Bendel

Because usually only one EK is used as the identity of a TPM, when this key is compromised, it is possible to forge the identity of the TPM and use it for attacks. It is also possible to recover the storage keys with techniques such as LPC bus sniffing.[15] Therefore, it is necessary to deploy an appropriate key and certificate revocation mechanism that requires additional effort from TPM vendors and device CAs. There is currently a gap in this area because of a lack of standards and support for it within the industry.

2.1.3.4 Enable Local Attestation When Needed at Boot-Time

Local attestation means that the booting process itself, rather than an external verifier, performs the verification before entering the next stage. This attestation mechanism is common for secure boot because the result of the code signature decides if the booting process can go to the next stage.

The first method for performing local attestation is to use a Dynamic Root of Trust for Measurement (DRTM) to verify an untrusted environment before moving further. In trusted boot, DRTM assumes that its initial state (preamble) is in an untrusted environment. Therefore, the boot process always needs to verify the signature of the combined code module (i.e., DCE module) to confirm that it is signed by the chipset manufacturer. If the signature verification fails, the platform then enters the remediation stage (see Section 5.2.6 in the TPM 2.0 architecture specification[16]). The remediation stage is manufacturer-specific and therefore can be implemented in various ways. According to the TCG D-RTM specification,[17] remediation shall ensure the process is terminated by either turning off or resetting a device.

The second method is to configure a local System Integrity Reference Repository (SIRR) for managing valid public keys for verifying the signed code blocks; this mechanism is the same that is used in secure boot. Automatic local attestation is also possible with the same command and can be used to check system integrity after the update to TPM firmware.

Note that with local attestation on typical general-purpose computer platforms, only parts of the software stack have been measured (i.e., through signature verification). For example, boot manager, boot policy, data events, etc., are validated only during remote attestation. Therefore, considering the threat against incomplete measurements, trusted boot must always perform remote attestation.

2.1.3.5 Always Use Remote Attestation after Boot-Time

Remote attestation means that there is an external entity that attests the TPM to make sure it is genuine and bonded with the reporting platform. This feature is specifically used with a TPM in trusted boot because the TPM stores the measurements that reflect the real running system information. In short, remote attestation should be able to answer two questions: (1) Is the TPM used genuine with a valid identity? (2) Does the system behave as expected?

The first question is answered through the use of EK pairs embedded inside a TPM. As described in Section 2.1.3.1, the EK is the identity of a TPM. Every TPM ships with a unique EK pair that is installed by the manufacturer. When attesting the identity of the TPM, the remote verifier asks for the attester's EK certificate and validates it against the corresponding CA certificate. If the EK certificate is valid, the verifier then knows the TPM is genuine and can therefore start to examine the TPM quote that includes the system measurement during the remote attestation protocol.

In addition to EK certificates, a TPM needs to issue an Attestation Identity Key Certificate that is bonded with the reported TPM quote. The reason for using an Attestation Identity Key Certificate together with the random nonce is to avoid a man-in-the-middle attack, in which attackers extract the valid measurement and reuse it for a compromised device. The verifier needs to validate this certificate and then use the public attestation key to decrypt the measurement. After obtaining the measurement value, the verifier can use a manufacturer's database to check the validity of the measurement. The procedure is depicted Figure 8.

Still, remote attestation has its limitations. The following section lists two examples of attacks (completeness attack and S3 resume attack) that can bypass remote attestation.

[15] "SySS Research Releases iCEstick FPGA Tool to Capture and Decrypt BitLocker Volume Keys," March 2020, Hackster.io—Gareth Halfacree
[16] Trusted Platform Module Library—Part 1: Architecture, Revision 01.59, Trusted Computing Group, November 2019
[17] TCG D-RTM Architecture, Version 1.0.0, Trusted Computing Group, June 2013

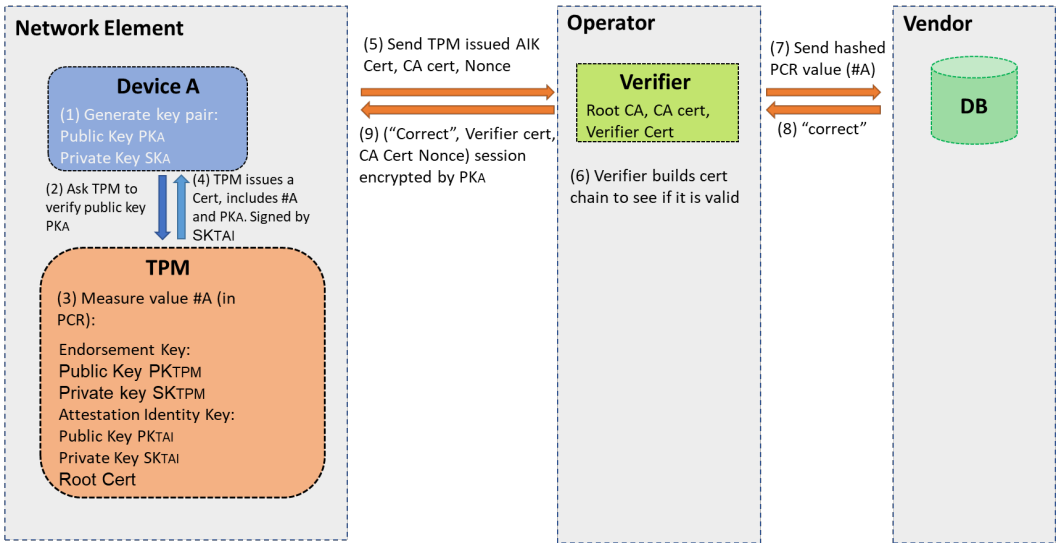

Figure 8: Remote Attestation Example for a Virtualized Device

2.1.3.6 Use Fresh and Comprehensive Information for Attestation

Completeness Attack

The completeness of measurement is the biggest challenge of trusted computing. Real implementations might not measure some of the firmware components and, therefore, miss an important measurement that extends the attack surface. Additionally, real implementations might not measure all the security-related settings. If some of the security capabilities were turned off through such settings, it would not be detected.

One example of attacking the completeness against trusted boot is Stuxnet. The virus Stuxnet usually accesses a supply chain network via an infected USB flash drive used by a person with physical access to the system. This virus travels across the network, scanning software on computers controlling a programmable logic controller (PLC), and then infects the rootkit on the PLC to modify the code and runs unexpected commands to the PLC while returning a loop of normal operation value feedback to the users.

S3 Resume Attack

To support power efficiency, a trusted platform may support sleep, also known as S3 (Stage 3) resume according to the Advanced Configuration and Power Interface (ACPI) specification. Before such a system goes to sleep or turns off, it needs to go to the S3 stage, where the current context including current system configuration, OS waking vector, and all Platform Configuration Register (PCR) values should be successfully saved. After the user wakes up the system, the boot process initializes the system according to the saved system configuration and jumps to the waking vector, continuing its operation.

During measurement, before a platform enters the S3 state, the OS sends TPM_Shutdown (STATE) to let the TPM save the context. When resuming from S3, BIOS sends the TPM_Startup (STATE) command to let the TPM restore the context. The TPM context includes all SRTM PCRs but does not include the DRTM PCRs.

One example of an S3 resume attack is Napper,[18] which attacks the consistency of measurements between stages to skip the TPM_Shutdown (STATE) command. Therefore, when the BIOS SRTM sends the TPM_Startup (STATE) command to the TPM, the TPM fails but the system may continue to boot after all the TPM PCR values are reset to zero. Attackers can firstly perform the normal measurement, extract it, and store into RAM. Then in the compromised state, attackers can power off the device to force the TPM to enter the S3 state. When the TPM wakes up, a hooked function will point the measurement to the stored normal measurement. Because the TPM PCR values are now zero and the system did not block the boot process after encountering an error, with the extended hash

[18] "Finally, I Can Sleep Tonight: Catching Sleep Mode Vulnerabilities of the TPM with Napper," Seunghun Han, Jun-Hyeok Park, Black Hat Asia 2019, March 26–29, Singapore

function, attackers can easily forge a normal PCR value and perform attacks. Remote attestation with these PCR values will show the system is trusted though the system has already been attacked.

In the scope of trusted boot, the remote attestation only assures the integrity of an installed software image, thus the importance of the completeness and freshness of measurements for the trusted boot process. The integrity of the attestor will impact the trustworthiness of the remote attestation, which requires the system to isolate the attestor and always measure it first before proceeding with the rest of the system measurements.

Ideally, the attestation mechanism should reflect the running system, not just the static software images. In situations where VMs or containers are running on the trusted platform, the system state is dynamic and may allow outside inputs at run-time, providing an opportunity for attackers to remotely attack the system without interrupting the booting process.

2.1.3.7 Complexity of Constrained Policies Implementations

Another traditional challenge for remote attestation is privacy. Especially when measurements apply to devices operated by users, the attestation process should make sure no sensitive data are sent to the verifier in case of possible insider attack or data leakage. To do so requires the remote verifier to verify the measurement without collecting any sensitive identity data from the machine, the use of a secure channel between attestor and verifier, and/or the deployment of privacy CAs to address privacy concerns. However, when dealing with devices such as cable modems or gateways, the running applications and their measurements rarely can be associated with specific individuals (same device models will carry the same measurements independently from the deployment) and, therefore, it is reasonable to adopt a simpler architecture that does not require masking attestation measurements.

3 Comparison

Because there is a great deal of complexity when it comes to the different secure boot approaches, it is common to confuse the definitions and distinctions of the different mechanisms. All of them have strong pros and cons, and it is important to approach them with an eye for the specific security needs for the ecosystem. To better understand the different deployment implications of the boot process, we focus on two specific definitions: secure boot and trusted boot (Figure 9). Within the scope of this paper, secured boot means the booting process follows one principle: “verify before execution.” Trusted boot, in short, is the combination of remote attestation and measured, which uses a combination of the mechanism and functionalities that measure each stage of the booting process. For the summary of the main differences between secure boot and trusted boot, refer to Table 1.

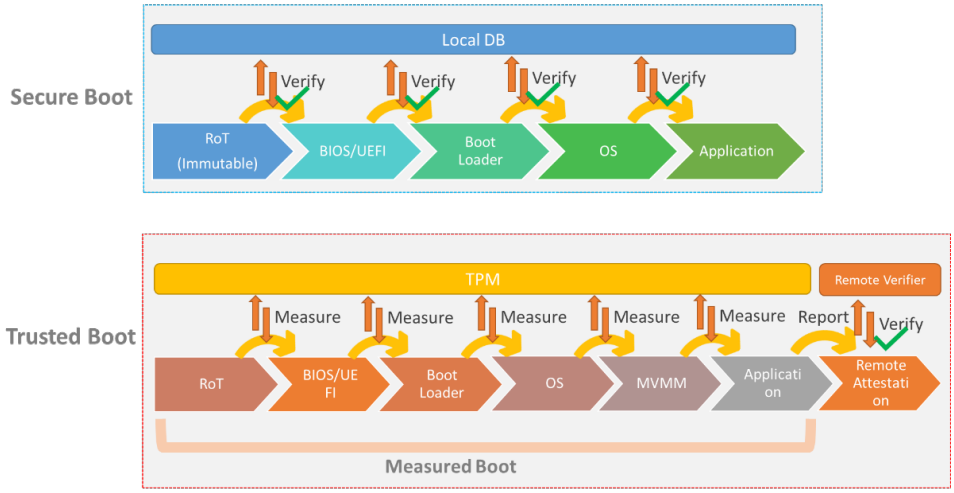

Figure 9: Comparison Between Secure Boot Workflow and Trusted Boot/Measured Boot Workflow

TABLE 1: SECURE BOOT VS. TRUSTED BOOT

	SECURE BOOT (WITH VERIFIED BOOT)	TRUSTED BOOT (WITH MEASURED BOOT)
Root of Trust	Root public key or root public key hash hardware-protected within a secure SoC.	Core root of trust (RoT) is the Trusted Platform Module (TPM); root storage keys and endorsement keys usually protected within the TPM and never go outside.
Local Attestation	Must be supported to verify the signature of the code piece before execution.	Optional; can be used in DRTM to verify the signature of the code piece before execution.
Remote Attestation	Does not require connectivity to any remote verifier to establish a chain of trust (CoT).	Needs to inspect TPM PCR values and TPM issued certificates with the remote verifier
Trusted Authority	An asymmetric signing key corresponding to a public key chain to the RoT protected in hardware. The public key is often included in a proprietary digital certificate (not X.509) in a chip vendor-specific format. The device OEM usually manages and protects this secure boot code signing key, although it could be managed by a third-party code signing service.	Pre-attestation is performed during installation, when the device certificate and TPM identity certificate must be verified separately by the remote verifier/Attestation CA. The TPM must exchange the attestation key with a remote verifier to obtain a certificate for this key. After booting, a remote verifier has to match the PCR measurements against their expected values for remote attestation to succeed.
Chain of trust	Established by each software layer verifying the next software layer before handing the control over (in the boot chain).	Established after going through the TPM Event Log and PCR values.
Accessibility	The device will become inaccessible if one of the software layers fails validation and no signed backup or recovery image is available in the non-volatile storage of the device. In that case, the booting process will halt, and the device may go into a reset.	The device with invalid system measurement can still be visible and accessible to operators after the boot up. Remote attestation is necessary for verifying the validity of the boot-up system measurement. Policy of network and services access should be configured based on the result of remote attestation to avoid the potential security risks.

4 Deployment Scenarios

The goal of this section is to provide considerations for trusted boot deployment in operator-managed networks. This includes embedded devices such as eDOCSIS or eMTA elements, constrained devices such as cable modems and managed gateways or access points, less constrained devices such as CMTS and RPDs, and general-purpose hardware such as management servers.

4.1 Customer Premises Equipment

In the cable network, the customer premises equipment includes many different types of devices and network equipment such as cable modems, DOCSIS gateways, eDOCSIS or eMTA elements, and set-up boxes. Customer premises equipment is usually customer-residential, usually connected to the coax, and integrated with the customer's home network. As a result, the customer has physical access to the devices and could potentially alter them without the operator's knowledge. The compromise of these devices can be very impactful for customers, as reliability, availability, and traffic privacy are critical to customers' user experiences.

Challenges occur based on all characteristics listed above, as well as criticality of costs, field deployment, user experience, etc. First, because these devices are usually customer-residential, no onsite technician is available for regular device checking, so the operation efficiencies such as local customer supports require extra costs. If the TPM is used in the deployed architecture, some may want devices to include a local attestation step to validate the measurements at each stage of the boot process. If the wrong components are loaded or there are TPM hardware reliability issues, the measurements might not be as expected and the boot process might fail. If TPM hardware reliability is a major concern, the suggested deployment path is to use a delayed measurement validation (remote attestation) after the boot process is completed. If deployed devices use hardware components such as an embedded TPM chip, it is very important to consider their reliability as malfunctioning TPMs can cause high support costs when considering the large number of customer premises equipment usually deployed in broadband networks.

Second, because customers have physical access to these devices, there might be some intentional or unwilling attacks/modifications that the TPM is not able to detect. For example, an attacker with physical access to the TPM chip can easily connect bus pins to an analyzer and decode the data to extract keys or perform an S3 resume attack.

Third, the capability of those devices needs to be considered when it comes to the risk of compromise. Some functionalities of TPMs such as remote attestation, although they require additional resources on the network, can help operators to prevent abuse at the core where compromises might affect large groups of users.

Currently, SoC vendors have already provided full secure boot capability. Cable modems, MTAs, and routers are often sold at retail where the cost of an additional TPM chip can make a big difference at being competitive. Realistically, a TPM processor should be included in the SoC before it becomes practical to utilize it for such devices.

4.2 Distributed Network Elements

In the cable network, distributed network elements include the RPD, RMD, spine, and leaf switches. The combination of these distributed network elements is known as the Converged Interconnect Network (CIN). By deploying distributed network elements, operators resolve historical reliability challenges by eliminating headend equipment and moving RF processing from the headend to the field. Configuration, provision, and the core network RF functionality have been extended to the edge network, and the redundancy is also ensured by node switching and splitting techniques. The boundary between the trusted core network and the untrusted edge network domain are now blurring.

Compared with secure boot, extra time is needed to perform a TPM self-test, system measurement, and remote attestation. Moreover, network availability is critical for performing the remote attestation process. If either network latency or operation time is critical for maintaining services, it is recommended to use secure boot for the distributed network elements and trusted boot for operator's servers in the back office. Notice that secure boot failure can cause a permanent denial-of-service attack if the boot process or the image is broken, though distributed network elements such as an RPD do not often perform software image updates. If a device fails at boot time, it can remain detached from the network without reporting the issue unless switching back to a back-up image. Therefore, the device should always support at least two software images. That said, if recovery from a backup software image also fails, operators do not know the real boot-up status of a device and thus need more time to locate the issue. The only way for operators to troubleshoot is to send a local technician for hands-on troubleshooting and run truck-rolls for these devices, where the physical locations may be far away from the primary office. Therefore, the Secure Software Download (SSD) procedure and code signing process are indispensable as proactive principles for secure boot.

Because of the distributed nature of such devices, it is even more important to cryptographically identify each distributed network element. A TPM chip can be used to protect the device's public key and certificate; the TPM's key management system and its tamper-resistant design increases the cost for an attacker to break the keys. The pre-attestation performed by vendors also ensures the device and its TPM chip are tied together. Moreover, the attestation identity certificates that issued for the device's TPM during the pre-attestation, coupled with the system boot status, can also be consecutively attested during the later remote attestation process to ensure the genuineness of the TPM and device identity. By contrast, secure boot can only perform local attestation and relies on the security of the device's non-volatile storage to secure the device's identity.

Because distributed network elements may cross typical operational group responsibility boundaries, the difficulties of deploying new techniques may impact various groups and require extra administration changes. Considering the fact that one distributed network element serves a larger number of subscribers, an automatic mechanism is preferred for monitoring, configuring, and attesting the boot process of such devices, in order to detect boot failures that might result in large scale service outages. For example, an administrator can configure a remote verifier to randomly pick up a group of distributed network elements and request for the system measurement. This remote verifier can also be configured to perform remote attestation after a period of time and access to a database for querying the validity of the received measurement. Unlike secure boot, operators are able to monitor the devices' status and configure the policy based on the remote attestation results, which helps to reduce hands-on troubleshooting and truck-rolls and increase the ability of managing the network accessibility of these devices.

4.3 Operator's Headend Equipment

In the cable network, an operator's headend equipment usually includes a CMTS/CCAP, MAC Manager, and AAA Server. Commonly, one type of equipment, such as a CMTS, needs to serve a large amount of connected network components at the same time and perform the functionalities towards both the operators' Network Management System (NMS) and the customer-facing network. Because this type of equipment usually allows configuration via SNMP or NETCONF from the network management system, the result of remote attestation and the validity of system measurements can be used to manage access to the NMS and to decide if such equipment is ready to serve or not.

In previous, more centralized types of architectures, because this type of device was deployed in an operator's headend, the security of the device was primarily provided by restricting physical access only to authenticated employees. In other words, the operator's headend equipment was always considered to be within the trusted domain. However, an insider attack or misconfiguration can still cause fatal failure for device boot up and function. For example, experienced technicians are usually required to access the CLI to troubleshoot or configure the device. Manual operations may result in accidental or willing misconfiguration. Therefore, it is necessary to use remote attestation to ensure that the secure configurations align with the operator's policy. Event logging is also required in order to audit such manual operations—any critical operations can be traced back to one person and, if error occurs, onsite technicians can use such information to locate the issue and decide the remediation approaches.

With those benefits, trusted boot seems to be a fit for an operator's headend equipment.

4.4 Provisioning Server System

In the cable network, the dedicated host environment comprises servers with specific functionalities in the operator's provisioning system (i.e., DHCP and TFTP servers).

The systems running on such servers have different options and varying support for boot-up security. For example, Ubuntu hosts support UEFI secure boot. In Gentoo Linux, trusted boot has been recommended, with both DRTM and SRTM supported. For other Linux systems such as Fedora, both UEFI secure boot and trusted boot can be configured by a system administrator.

Mac hosts with an Apple T2 security chip perform a secure boot from the Boot ROM. When an Apple device is turned on, the T2 chip will verify the signature of the iBoot bootloader, kernel, kernel extension code, and UEFI firmware on the T2 chip. On the other hand, Mac computers without an Apple T2 security chip do not support secure boot, which means UEFI firmware loads the bootloader without verification. In this case, System Integrity Protection (SIP), FileVault, and Firmware Password can be enabled to protect malicious writes of bootloader and kernel within the running macOS.

For Windows hosts, both secure boot and trusted boot have been used during the booting process. Though secure boot is used to verify the UEFI firmware and provides the root of trust for the whole boot-up process, the trusted boot process leverages TPM chips to perform system measurements for OS loader, kernel, drivers, and system files. Depending on the implementation and configuration needs, non-Microsoft tools can be used to implement a remote attestation client and trusted attestation server for verifying the measurement.

In conclusion, for a dedicated host environment, both secure boot and trusted boot can be configured to deliver boot-time security. The common practice is to use UEFI secure boot for local attestation with signature verification. If trusted boot is deployed, remote attestation can be enabled through manual configuration with third-party toolkits,[19] and extra effort is needed for deploying a trusted attestation server and client. Remote attestation and identity revocation are still the gaps in deploying trusted boot in such an environment.

4.5 Virtualized Environment

Virtualization is a technology that uses software called a hypervisor to abstract the machine's resources and manage multiple running virtual environments called virtual machines (VMs). Virtualization allows an organization to run multiple VMs on a single physical computer hardware, which helps to reduce downtime and enhance resiliency as well as increase efficiency and productivity for operators.

One challenge for the virtualized environment is linking the VMs with physical TPM identities. In a virtualized environment, multiple VMs share the hardware resources. When performing the remote attestation, the measurement of each VM should be encrypted with a unique attestation certificate issued by a virtualized TPM (v-TPM) without interfering with others.

In the real implementation, there are two different types of Root of Trust for Measurement (RTM): Static RTM (SRTM) and Dynamic RTM (DRTM) (Figure 10).

[19] "Remote Attestation with Tpm2 Tools," June 2020, tpm2-software community—Imran Desai

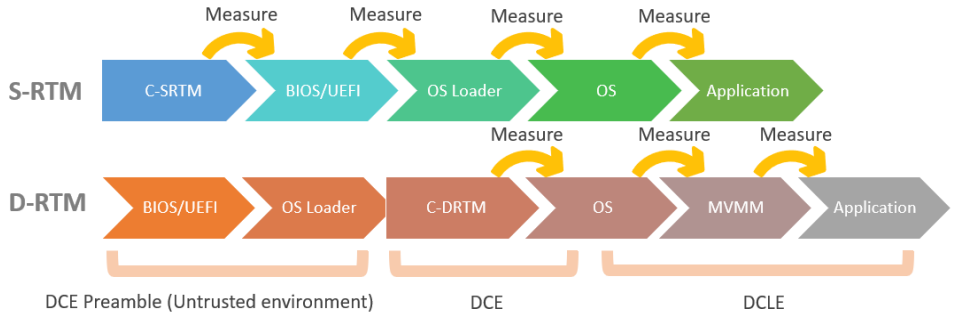

Figure 10: SRTM Flow versus DRTM Flow

When the device uses the Static RTM (SRTM), the initial trust state of the CoT is created after the power-on reset and cannot be changed. The biggest problem of SRTM, however, is that the boot process must make sure every required component is measured. One missing measurement can cause a broken CoT, resulting in boot process failure. That said, the SRTM process always requires the device to restart in order to perform new measurements. In a virtualized environment, if one VM fails to boot up or has invalid measurement, the whole machine needs to be reboot in order to establish a new CoT for attestation. Therefore, a SRTM is not applicable to a virtualized environment where isolation of VMs matters. To resolve this problem, TCG introduced the Dynamic RTM (DRTM), with the initial trust state in the middle of the platform boot.

Neither secure boot nor the SRTM type of trusted boot is applicable to virtualization because they always require a reboot to establish the CoT. The DRTM, as described above, can establish the CoT in the middle of the booting process and launches a measured virtual machine monitor (MVMM) to monitor running applications. If one VM fails, the CoT can still be established without rebooting the devices, which ensures the availability of other VMs on the same physical device.

For attestation, instead of using a physical RoT, TCG introduced the vTPM to report the status of the hypervisor and VMs to a remote verifier. That is, each VM can leverage the virtual trusted platform instance with the vTPM, which includes security capabilities such as virtual Root of Trust for Measurement (vRTM), virtual root of trust for reporting (vRTR), or virtual root of trust for storage (vRTS). More details can be found in TCG's Virtualized Trusted Platform Architecture specification[20] and NIST's recent draft on hardware-enabled security.[21]

Conclusion

This paper discussed the mechanism, security principles, and deployment scenarios for both secure boot and trusted boot and the ways they can be used to secure the boot process in cable access networks. When comparing secure boot and trusted boot, one needs to address deployment complexity, integration, and cost considerations. Specifically, because of the added infrastructure requirements of trusted boot, secure boot might be a cheaper deployment option for edge devices. From this point of view, especially in early-adoption phases, it is recommended to implement the secure boot for customer premises equipment where device cost is a critical aspect for manufacturers and operators. For larger capacity devices such as RPDs, CCAP cores, or dedicated hosts, trusted boot solutions could be the key to lowering the risk of compromise in core networks. In the future, given the expected reduction in deployment costs and increased experiences with the technology, we estimate that it will be possible to implement trusted boot processes efficiently for edge devices as well.

[20] Virtualized Trusted Platform Architecture Specification, Version 1.0, Revision 0.26, Trusted Computing Group, September 2011

[21] "Hardware-Enabled Security: Enabling a Layered Approach to Platform Security for Cloud and Edge Computing Use Cases," NISTIR 8320 (Draft), National Institute of Standards and Technology, May 2021