Securing the Boot Process

THE HARDWARE ROOT OF TRUST

JESSIE FRAZELLE

The boot sequence for a machine typically starts with the BMC (baseboard management controller) or PCH (platform controller hub). In the case of an Intel CPU, the Intel Management Engine runs in the PCH and starts before the CPU. After configuring the machine's hardware, the BMC (or PCH, depending on the system) allows the CPU to come out of reset. The CPU then loads the boot firmware (or UEFI, unified extensible firmware interface) from the boot firmware SPI (Serial Peripheral Interface) flash. The boot firmware then accesses the boot sector on the machine's persistent storage and loads the bootloader into the system memory. It then passes execution control to the bootloader, which loads the initial operating system image from storage into system memory and passes execution control to the operating system. For example, in popular Linux distros, GRUB (Grand Unified Bootloader) acts as the bootloader and loads the operating system image for the machine.

This is much like a relay race where one team member passes a baton to another to win the race. In a relay race, you hopefully know the members of your team and trust them to do their part for the team to get to the finish line. With machines, trust is a bit more complex. How can you verify that each step in the boot sequence is running software that you know is secure? If our hardware or software has been compromised at any point in the boot

sequence then the attacker has the most privilege on our system and likely can do anything they want.

The goal of a hardware root of trust is to verify that the software installed in every component of the hardware is the software that was intended. This way you can verify and know without a doubt whether a machine's hardware or software has been hacked or overwritten by an adversary. In a world of modchips[16], supply chain attacks, evil maid attacks[7], cloud provider vulnerabilities in hardware components[2], and other attack vectors it has become more and more necessary to ensure hardware and software integrity.

This article is an introduction to a complicated topic; some sections just touch the surface, but the intention is to provide a full picture of the world of secure booting mechanisms.

TRUSTED PLATFORM MODULE

A TPM (trusted platform module) is a standard for a dedicated microchip designed to secure hardware through integrated cryptographic keys. TPM was standardized by the ISO (International Organization for Standardization) and the IEC (International Electrotechnical Commission) in 2009 as ISO/IEC 11889.[9] The TPM is typically installed on the motherboard of a computer, and it communicates with the remainder of the system using a hardware bus.

A TPM has the following features:[18]

- A random number generator
- A way to generate cryptographic keys
- Integrity measurement

- Attestation
- Wrapping/binding keys
- Sealing/unsealing keys

Integrity measurement

Measurement is the process through which information about the software, hardware, and configuration of a system is collected and digested. At load time, the TPM uses a hash function to fingerprint an executable and its configuration. These hash values are used in attestation to reliably establish code identity to remote or local verifiers. The hash values can also be used in conjunction with the sealed storage feature. A secret can be sealed along with a list of hash values of programs that are allowed to unseal the secret. This allows the creation of data files that can be opened only by specific applications.

Attestation

Attestation reports the state of the hardware and software configuration. The integrity measurement software in charge of creating the hash key used for the configuration data determines the extent of the summary. The goal of attestation is to prove to a third party that your operating system and application software are intact and trustworthy. The verifier trusts that attestation data is accurate because it is signed by a TPM whose key is certified by the certificate authority (CA). TPMs are manufactured with a public/private key pair built into the hardware, known as the endorsement key. The endorsement key is unique to a specific TPM and is

signed by a trusted CA. The trust for attestation data is dependent on the trust for the CA that originally signed the endorsement key.

Attestation can reliably tell a verifier which applications are running on a client machine, but the verifier must still make the judgment about whether each given piece of software is trustworthy.

When booting a machine securely, you want the first instruction run on that machine to be the one you would expect to run.

Wrapping/binding a key

A machine that uses a TPM can create cryptographic keys and encrypt them so that they can be decrypted only by the TPM. This process, known as *wrapping* or *binding* a key, can help protect the key from disclosure. Each TPM has a master wrapping key, also known as the storage root key, which is stored within the TPM. The private portion of a storage root key or endorsement key that is created in a TPM is never exposed to any other device, process, application, software, or user.

Sealing/unsealing a key

A machine that uses a TPM can also create a key that has not only been wrapped, but is also tied to certain platform measurements. This type of key can be unwrapped only when those platform measurements have the same values that they had when the key was created. This process is known as *sealing* the key to the TPM. Decrypting the key is called *unsealing*. The TPM can also seal and unseal data that is generated outside the TPM. With this sealed key and software you can lock data until specific hardware or software conditions are met.

CUSTOM SILICON

It is important to note the limitations of TPMs and provide some solutions to those. TPMs can attest that the firmware running on a machine is the firmware the user wants to run, but there is no mechanism in a TPM for verifying that the code is secure. It is up to the user to verify the security of the firmware and to ensure it does not contain any backdoors, which is impossible if the code is proprietary.

When booting a machine securely, you want the first instruction run on that machine to be the one you would expect to run. A TPM is insufficient for verifying that the actual bits of code to be executed are secure, so a few companies have created their own silicon for expanding on the security of TPMs.

Google's Titan

For Google's infrastructure, as well as Chromebooks, Google expanded on the security of the TPM with its own chip, Titan. Google open sourced[5] a version of Titan[9] (with both specs and code), which is under active development, in October 2019. In creating Titan, Google added two new features that did not exist in TPMs: first-instruction integrity and remediation.

First-instruction integrity

First-instruction integrity allows verification of the earliest code that runs on each machine's startup cycle. Titan observes every byte of boot firmware by interposing itself between the boot firmware flash (BIOS) of the BMC (or PCH) and the main CPU via the SPI bus. Therefore, the

boot sequence for a machine with a Titan chip is different from a normal boot sequence.

The boot sequence with Titan is as follows:

1. Titan holds the machine in reset.
2. Titan's application processor executes code from its embedded read-only memory (boot ROM).
3. Titan runs a memory built-in self-test to ensure that all memory (including ROM) has not been tampered with.
4. Titan verifies its own firmware using public-key cryptography, and mixes the identity of this verified code into Titan's key hierarchy.
5. Titan loads the verified firmware.
6. Titan verifies the host's boot firmware flash (BIOS/UEFI).
7. Titan signals readiness to release the rest of the machine from reset.
8. The CPU loads the basic firmware (BIOS/UEFI) from the boot firmware flash, which performs further hardware/software configuration.
9. The rest of the standard boot sequence continues.

Holding the machine in reset while Titan cryptographically verifies the boot firmware, Titan enables the verification of the first instruction. Titan knows what boot firmware and operating system booted on your machine from the very first instruction. In fact, you even know which microcode patches may have been fetched before the boot firmware's first instruction.

Remediation

What happens when we need to patch bugs in Titan's firmware? This is where remediation comes into play. In the

event of patching bugs in the Titan firmware, trust can be reestablished through remediation. Remediation is based on a strong cryptographic identity. To provide a strong identity, the Titan chip manufacturing process generates unique keying material for each chip. The Titan-based identity system verifies not only the provenance of the chips creating the CSRs (certificate signing requests), but also the firmware running on the chips, as the code identity of the firmware is hashed into the on-chip key hierarchy. This property allows Google to fix bugs in Titan firmware and issue certificates that can be wielded only by patched Titan chips.

The Titan-based identity system enables back-end systems to securely provision secrets and keys to individual Titan-enabled machines or jobs running on those machines. Titan is also able to chain and sign critical audit logs, making those logs tamper-evident. This ensures that audit logs cannot be altered or deleted without detection, even by insiders with root access to the relevant machine.

Microsoft's Cerberus

Microsoft open sourced[11] the specs for its chip, Cerberus[11] (at the time of writing this article, only the specs have been open sourced). Like Titan, Cerberus interposes on the SPI bus where firmware is stored for the CPU. This allows Cerberus to continuously measure and attest these accesses to ensure firmware integrity and thereby protect against unauthorized access and malicious updates.

Apple's T2

Apple is a poster child for secure booting devices. Most

people remember when the FBI wanted a backdoor into iPhones and Tim Cook refused.[5] Between Macs, iPhones, and Chromebooks, an industry standard for products includes security by default.

For Apple machines, secure boot is done with Apple's T2 chip.[1] Ivan Krstic of Apple gave a talk at Black Hat 2019[12] detailing the boot process for a Mac with Apple's T2 chip.

Apple's requirements for T2 were:

- Signature verification of the complete boot chain.
- System software authorization (server-side downgrade protection).
- Authorization "personalized" for the requesting device (not portable).
- User authentication required to downgrade secure boot policy.
- Secure boot policy protected against physical tampering.
- System can always be restored to a known-good state.

The boot sequence for a machine using a T2 chip is as follows:

- The machine is powered on.
- T2 ROM is loaded and executed.
- T2 ROM passes off to iBoot, the bootloader.
- The bootloader executes the bridgeOS kernel, the kernel for the T2 chip.
- The bridgeOS kernel passes off to the UEFI firmware for the T2 chip.
- The T2 chip then allows the CPU out of reset and loads the UEFI firmware for the CPU.
- The UEFI firmware for the CPU then loads macOS booter.

- The macOS booter then executes the macOS kernel.

One important design element of the T2 chip is how Apple verifies the version of MacOS running on a computer. T2 verifies the hash of MacOS against a list of approved hashes for running. Apple is in a unique position to have this level of verification since they own the entire stack and prevent users from running any other OS on their devices. If you would like to go deeper into the internals of the T2 chip, read the slides for Ivan Krstic's Black Hat talk.[12]

PLATFORM FIRMWARE RESILIENCY

Chip vendors are investing in PFR (platform firmware resiliency) based on NIST (National Institute of Standards and Technology) guidelines.[15] These guidelines focus on ensuring the firmware remains in a state of integrity, detecting when it has been corrupted, and recovering the pieces of firmware back to a state of integrity.

PFR addresses the vulnerability of enterprise servers that contain multiple processing components, each having its own firmware. This firmware can be attacked by hackers who may surreptitiously install malicious code in a component's flash memory that hides from standard system-level detection methods and leaves the system permanently compromised.

The PFR specification is based on the following principles:

- Protection, ensuring that firmware code and critical data remain in a state of integrity and are protected from corruption, such as the process for ensuring the authenticity and integrity of firmware updates.
- Detection, detecting when firmware code and critical

data have been corrupted.

- Recovery, restoring firmware code and critical data to a state of integrity in the event that any such firmware code or critical data are detected to have been corrupted, or when forced to recover through an authorized mechanism.

Vendors have been building features around the NIST guidelines for PFR. Intel[8] and Lattice Semiconductors[13] each has such a product.

UEFI SECURE BOOT

UEFI Secure Boot[21] is designed to ensure that EFI binaries that are executed during boot are verified, either through a checksum or a valid signature, backed by a locally trusted certificate. When a machine using UEFI Secure Boot powers on, the UEFI firmware validates that each EFI binary either has a valid signature or the binary's checksum is present on an allowed list. Counter to the allow list is a deny list that is also checked to ensure no binary's checksum or signature exists on it. Users can configure the list of trusted certificates and checksums as EFI variables. These variables get stored in non-volatile memory used by the UEFI firmware environment to store settings and configuration data.

The UEFI kernel is extremely complex and has millions of lines of code. It consists of boot services and runtime services. The specification[19] is quite verbose and complex. The UEFI kernel is a common vector for many vulnerabilities since it has some of the same proprietary code used on many different platforms. The UEFI kernel is shared on multiple platforms, making it a great target for attackers. Additionally, since only UEFI can rewrite

itself, exploits can be made persistent. This is because UEFI lives in the processor's firmware, typically stored in the SPI flash. Even if a user were to wipe the entire operating system or install a new hard drive, an attack would persist in the SPI flash.

INTEL'S BOOT GUARD

Boot Guard is Intel's solution to verify the firmware signatures for the processor. Boot Guard works by flashing the public key of the BIOS signature into the field programmable fuses (FPFs), a one-time programmable memory inside Intel Management Engine (ME), during the manufacturing process. The machine then has the public key of the BIOS and it can verify the correct signature during every subsequent boot. However, once Boot Guard is enabled by the manufacturer, it cannot be disabled.

The problem with Boot Guard is that only Intel or the manufacturer has the keys for signing firmware packages. This makes it impossible to use coreboot, LinuxBoot, or any other equivalents as firmware on those processors. If you tried, the firmware would not be signed with the correct key, and the failed attempt to boot would brick the board.

Matthew Garrett wrote a great post about Boot Guard that highlights the importance of user freedom when it comes to firmware[4]. The owner of the hardware has a right to own the firmware as well. Boot Guard prevents this. In the security keynote at the 2018 Open Source Firmware Conference[6], Trammel Hudson described how he found a vulnerability to bypass Boot Guard, CVE-2018-12169[3]. The bug[20] allows an attacker to use unsigned firmware and boot normally, completely negating the purpose of Boot

Guard. Because Boot Guard is tied to the CPU, it does not have the control that a custom silicon hardware root of trust has when it comes to other firmware for components in the system.

SYSTEM TRANSPARENCY

The Mullvad virtual private network service published a paper on what it calls system transparency,[17] which is aimed at facilitating trust for the components of a system by giving every server a unique identity, limiting the attack surface and mutable state in the firmware and allowing both owners and users to verify all software running on a platform starting from the first instruction executed after power on.

ST accomplishes these goals by following seven principles:

1. Identity binding. A key ceremony of each server to bind the server's unique identity with a difficult to forge physical artifact like a video.
2. Physical write-protection of the firmware. Writable code sections are a mutable state, so System Transparency limits the possible changes to this critical piece of code. Read-only code also serves as a root of trust for all other software-enforced security mechanisms.
3. Tamper detection. Attackers cannot be stopped from changing the content of the firmware flash by replacing the actual chip. So, violations of the physical integrity of the server hardware need to be detectable.
4. Measured boot. System Transparency has the goal to give all parties insight into what code was run as part of the system boot. A measured boot in combination with remote attestation allows third parties to acquire a

cryptographic log of the boot.

5. Reproducible builds. Ensures that if a binary artifact is built once, it can be built again and again and produce the same artifact. This establishes a verifiable link between the human-readable code and the binary that was attested using the measured boot mechanism.

6. Immutable infrastructure. System transparency only works when changes to the operating system are limited. Allowing somebody to log into the system and make arbitrary changes invalidates all guarantees of a measured boot.

7. Binary transparency log. All firmware and OS images that can be booted on a system are signed by the system's owner and are inserted into a public, append-only log. Users of the system can monitor this log for new entries and catch malicious system owners booting backdoored firmware on new servers.

Related articles

Security for the Modern Age
Securely running processes that require the entire syscall interface
Jessie Frazelle
https://queue.acm.org/detail.cfm?id=3301253

Simulators: Virtual Machines of the Past (and Future)
Has the time come to kiss that old iron goodbye?
Bob Supnik
https://queue.acm.org/detail.cfm?id=1017002

Automating Software Failure Reporting
We can only fix those bugs we know about.
Brendan Murphy
https://queue.acm.org/detail.cfm?id=1036498

THE IMPORTANCE OF OPEN SOURCE FIRMWARE

Securing the boot process with a hardware root of trust has various implementations throughout the industry. Without open source firmware, the proprietary bits of the boot process still lack the visibility and auditability to ensure

that software is secure. Even if you can verify through a hardware root of trust that the hash of proprietary firmware is the hash known to be true, you need visibility into the firmware's source code for assurance it does not contain any backdoors. Through this visibility you can also gain ease of use in debugging and fixing problems without relying on a vendor.

Firmware is scattered throughout motherboards of machines and their components; it is in the CPU (central processing unit), NIC (network interface controller), SSD (solid-state drive), HDD (hard-disk drive), GPU (graphics processing unit), fans, and more. To ensure the integrity of a machine, all these components must be verified. In the future, these custom silicon chips will interpose not only on the SPI flash but also on every other device communicating with the BMC.

If you would like to help with the open source firmware movement, push back on your vendors and the platforms you are using to make their firmware open source.

Acknowledgments

Thank you to Ivan Krstic, Matthew Garrett, Kai Michaelis, Fredrik Strömberg, and Trammell Hudson for all their research and work in this area, which helped me write this article.

References

1. Apple. 2018. Apple T2 Security Chip; https://www.apple.com/mac/docs/Apple_T2_Security_Chip_Overview.pdf
2. Cimpanu, C. Hackers can hijack bare-metal cloud servers by corrupting their BMC firmware; https://

www.zdnet.com/article/hackers-can-hijack-bare-metal-cloud-servers-by-corrupting-their-bmc-firmware/
3. Common Vulnerabilities and Exposures. 2018; https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-12169
4. Garrett, M. 2015. Intel Boot Guard, Coreboot and user freedom; https://mjg59.dreamwidth.org/33981.html
5. Google Open Source Blog. 2019. OpenTitan—open sourcing transparent, trustworthy, and secure silicon; https://opensource.googleblog.com/2019/11/opentitan-open-sourcing-transparent.html.
6. Hudson, T. 2018. Open Source Firmware Conference's Security Keynote; https://trmm.net/OSFC_2018_Security_keynote#Boot_Guard
7. Hudson, T. Thunderstrike EFI bootkit FAQ; https://trmm.net/Thunderstrike_FAQ#Does_anyone_actually_use_evil-maid_attacks.3F
8. Intel. 2017. Intel Data Center Block with Firmware Resilience; https://www.intel.com/content/dam/www/public/us/en/documents/solution-briefs/firmware-resilience-blocks-solution-brief.pdf.
9. ISO/IEC 11889-1:2009. Information technology—trusted platform module; https://www.iso.org/standard/50970.html.
10. Kahney, L. 2019. The FBI wanted a back door to the iPhone. Tim Cook said no. Wired (April 16); https://www.wired.com/story/the-time-tim-cook-stood-his-ground-against-fbi/.
11. Kelly, B. 2017. Open Compute Project—Project Cerberus Security Architecture Overview Specification; https://github.com/opencomputeproject/Project_Olympus/

blob/master/Project_Cerberus/Project%20Cerberus%20Architecture%20Overview.pdf.
12. Krstic, I. 2019. Behind the scenes of iOS and Mac security; https://i.blackhat.com/USA-19/Thursday/us-19-Krstic-Behind-The-Scenes-Of-IOS-And-Mas-Security.pdf.
13. Lattice Semiconductors. Universal Platform Firmware Resiliency (PFR)—Servers; http://www.latticesemi.com/Solutions/Solutions/SolutionsDetails02/PFR.
14. OpenTitan. 2019. Introduction to OpenTitan; https://docs.opentitan.org/.
15. Regenscheid, A. 2018. Platform firmware resiliency guidelines. NIST Special Publication 800-193; https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-193.pdf.
16. Robertson, J., Riley, M. 2018. The Big Hack: How China Used a Tiny Chip to Infiltrate U.S. Companies; https://www.bloomberg.com/news/features/2018-10-04/the-big-hack-how-china-used-a-tiny-chip-to-infiltrate-america-s-top-companies
17. Strömberg, F. 2019. System transparency; https://mullvad.net/media/system-transparency-rev5.pdf.
18. Trusted Computing Group. 2011. TPM main, part 1, design principles; https://trustedcomputinggroup.org/wp-content/uploads/TPM-Main-Part-1-Design-Principles_v1.2_rev116_01032011.pdf.
19. UEFI. https://uefi.org/specifications
20. Wang, Jian. 2019. Bug 1614 (CVE-2019-11098) - BootGuard TOCTOU vulnerability; https://bugzilla.tianocore.org/show_bug.cgi?id=1614
21. Wilkins, R. 2013. UEFI SECURE BOOT IN MODERN COMPUTER SECURITY SOLUTIONS ; https://uefi.org/

sites/default/files/resources/UEFI_Secure_Boot_in_Modern_Computer_Security_Solutions_2013.pdf

Jessie Frazelle *is the co-founder and Chief Product Officer of the Oxide Computer Company. Before that, she worked on various parts of Linux including containers and also the Go programming language.*