CHAPTER 6

Boot with Integrity, or Don't Boot

You can't build a great building on a weak foundation. You must have a solid foundation if you're going to have a strong superstructure.

—Gordon B. Hinckley

You are on a business trip and staying in a nice hotel. You leave your laptop in the room while going out for a dinner appointment. The laptop has its full disk-encryption feature enabled. Being reasonably paranoid, you even turned off the laptop. You believe that the laptop and your confidential files stored in it are safe and secure. However, that may not be true. An "evil maid" who can physically access the laptop on the sly for just two minutes may be able to steal your drive encryption password without a trace. Consequently, the confidentiality of all encrypted data on the laptop is in danger.

How does the evil maid do it? The trick is the boot process. End-to-end security is essential. The boot security is as critical as, if not more critical than, runtime security.

For the past decade, the effort of securing computers has been focused largely on mitigating *runtime* threats. Numerous solutions have been developed to safeguard the integrity of computer systems and protect users' assets. These solutions include but are not limited to antivirus, network firewalls, and password managers. Some of these solutions are software-based; others are either dedicated hardware devices or hybrid designs made up of software and hardware. Most of these solutions mean to thwart certain types of security threats at runtime of the system. Drive encryption programs including TrueCrypt, PGP, and BitLocker) adopt a preboot authentication that is launched during the boot process as an extension of the BIOS before the operating system (such as Windows, Linux, Android, iOS, and so forth) is loaded.

The problem is the lack of end-to-end protection. Most software solutions are available only after being loaded by the operating system. In other words, during the boot process—that is, from the moment a user presses the power button to when the operating system takes control and finishes loading the security solutions—the computer is not benefiting from the services offered by the security measures and is hence vulnerable. Drive encryption schemes that start during the boot do not depend on the operating system to function, but they do rely on the integrity of the boot loader that loads them.

Admittedly, runtime protection is pivotal. The amount of time a computer typically spends on boot today is fairly small compared to how long the operating system is running. Operating systems have extensive interfaces and connectivity that make the attack surface wide and open. In contrast, the boot is a relatively short and contained process. As a result, attacks against the boot are more difficult to mount and succeed.

But a building is only as strong as its foundation. Hacking a computer's boot loader is similar to replacing a mansion's concrete foundation with sand. The components that are involved in the boot process comprise the root of trust for the entire system. A compromised boot loader renders the operating system—and all programs running on it—untrustworthy, including the antivirus, firewalls, and even drive encryption utilities.

Boot Attack

The boot process and components participating in the process vary, depending on the architecture of the system. How a computer boots today is significantly different and more complex than it was a decade ago. At a high level, most computers follow the boot sequence shown in Figure 6-1.

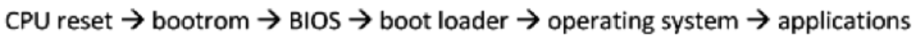
CPU reset → bootrom → BIOS → boot loader → operating system → applications

Figure 6-1. *Boot flow*

The BIOS (basic input/output system) is a firmware component stored in nonvolatile memory, usually a flash chip. The BIOS loads the boot loader, which is the first software component loaded during the boot process. The boot loader is stored in the hard drive, together with the operating system and applications.

For attackers, it is preferable to compromise a component that is loaded earlier than one loaded later, because taking control at an early stage enables control over all subsequent components. Successful attacks against user-mode software programs may not be glorious accomplishments in the security community nowadays. Instead, the BIOS and boot loader are becoming more interesting targets. A number of such attacks were published in the recent years. Here are two examples:

- *Attacking BIOS*: This type of attack replaces an authentic BIOS with an attacker's BIOS that contains malicious code. There have been attacks against the UEFI (Unified Extensible Firmware Interface) secure boot.
- *Attacking boot loader*: This type of attack usually installs a boot kit (a variant of root kits that runs in the kernel mode) under an attacker's control that infects the boot loader. The boot kit can be used to steal secrets during the boot path; for example, logging the user's drive encryption password.

If an adversary manages to modify the BIOS or boot loader code without authorization, then a straightforward damage he can realize is to corrupt the BIOS or boot loader and render the computer unbootable and inoperable (this category of attack is called *bricking*). The most famous example of this kind is the CIH virus, which resulted in reportedly millions of computers failing to boot in the late 1990s. The CIH virus, named after its author Chen Ing-Hau, a student at Taiwan's Tatung University, flashes and rewrites the BIOS region with junk so the infected computers can no longer start. Generally speaking, bricking the attacker's own device yields no benefits to the attacker. But if such bricking attacks can be mounted remotely and widely spread with viruses like CIH, it will cause substantial monetary loss.

In today's operating system, writing to flash or a boot loader without physical access is an incredibly privileged operation and hence more difficult to implement than 20 years ago. The bricking attacks against the boot path cause little or no harm on newer computers that are shipped with backup BIOS images on the flash and recoverable boot loaders on a special region of the hard drive or from the manufacturer-provided recovery disc. Most reputable antivirus utilities are capable of monitoring the integrity of the boot loader and of killing viruses that infect the boot loader. Pure bricking attacks against the boot path are considered out of scope in the remainder of this chapter.

Evil Maid

Joanna Rutkowska of the Invisible Things Lab was the first to describe the "Evil Maid" attack[1] in October 2009. In the Evil Maid attack, the maid attacker boots the victim's unattended laptop with her USB stick, which contains a bootable and stripped Linux operating system. The USB stick then uses the POSIX command `dd` to install a malicious boot kit, which changes the legitimate boot loader with a hook for recognizing and recording the full drive encryption passphrase later when the victim turns on his laptop and types in the passphrase on the keyboard. The malicious boot kit also recalculates certain fields of the MBR (master boot record), including the boot loader hash and size, in order to make it look like a legitimate MBR. The recorded passphrase is stored on the hard drive and it can be sent over the network to the attacker, or simply be retrieved by the evil maid the next day, when she can access the laptop and boot to her USB stick again. Once the encryption passphrase is acquired, the maid can just clone the victim's encrypted drive so she can steal all data on it.

Notice that the Evil Maid attack works only on a laptop that is turned off, because the attack takes advantage of the lack of boot integrity protection, and the drive encryption passphrase is entered by the user only during boot. If the maid deliberately turned off a sleeping or hibernating computer in order to mount her attack, then the victim would notice that something was wrong and suspect that someone had done something to his laptop. However, why would the victim power off his laptop in the first place, while he is going out for just an hour for dinner? The average user may not do so.

As a matter of fact, a paranoid professional user who has heard of the "cold boot" attack[2] may actually turn off his laptop even if he will be away for a short time. The researchers that presented the cold boot attack reports found that, based on experiments, the DRAM (dynamic random-access memory) still retains its content within a certain amount of time after the power is removed, even at the room temperature. Colder environments prolong the duration of the memory remanence. This observation is contrary to the popular assumption that DRAM would lose its data almost instantly when not being refreshed. The time period for which data resides in DRAM after power removal is generally long enough for an experienced attacker to figure out the drive encryption key from the DRAM. To counter such attacks, it is advisable to power down a laptop before leaving it unattended.

As you can see from the scenarios of the Evil Maid attack, without boot integrity protection, drive encryption techniques are able to safeguard your data only for cases where a thief steals and possesses your computer for good and attempts to retrieve plaintext data from it. If an attacker can secretly and physically access your computer for some period of time without you knowing, and then return it back to you, then the drive encryption cannot protect your data. This is not the fault of any specific drive encryption solution, but the limitation of the technology defined by its security model. The Evil Maid attack is simply out of scope if the user temporarily gives up the physical control of his laptop, that is, this scenario is not something that the encryption itself is intended or capable to mitigate.

To address this loophole, the security protection must start from the very beginning and cover the entire boot process. If the boot path is secured on the platform, then an evil maid will not be able to easily alter the MBR, so full drive encryption schemes can survive the attack.

BIOS and UEFI

The BIOS is the first piece of firmware that executes upon computer power-on. It is stored in nonvolatile memory, such as a flash chip on the motherboard. The fundamental functionality of the BIOS firmware is to initialize and self-test low-level hardware components of the computer, such as the CPU, keyboard, display, DRAM, and so forth, as well as to load the boot loader for the operating system from the hard drive. For a system with the security and management engine enabled, the BIOS is also responsible for communicating with the engine for basic configuration and reserving a predefined size of DRAM for the engine's dedicated access.

In fact, the BIOS is a standard that defines the platform firmware interface to the operating system. The term BIOS also refers to the firmware that implements the standard. In recent years, the UEFI standard[3] has been replacing the conventional BIOS standard, which has several limitations (such as a 16-bit real mode and a 1MB addressable memory) that are posing difficulty in meeting the needs of modern computers. Like the BIOS, the UEFI specification defines an interface between the operating system and the platform firmware, and the interface is designed to communicate only necessary information in order for the operating system to start. Besides supporting larger memory and a disk boot, the UEFI also introduces useful add-on features such as secure boot. Notice that the UEFI is backward-compatible with the BIOS standard. In this chapter, the term *BIOS* refers to the platform firmware that runs at boot, which may be either a conventional BIOS or a UEFI-compatible one.

Everything starts with BIOS on a computer, including security. If the BIOS is compromised, then all security countermeasures deployed after BIOS are essentially at risk. The era of the CIH virus—when a Windows application could program the flash and corrupt the BIOS—is long gone. Nevertheless, security researchers have reported BIOS alteration attacks using advanced techniques in recent years.

BIOS Alteration

At the Black Hat Europe conference in 2006, John Heasman presented a rootkit made possible by altering BIOS's ACPI (advanced configuration and power interface) table[4] The rootkit can infect Windows during Windows installation. This attack requires the capability of reflashing the flash chip where the BIOS is stored. At the 2009 CanSecWest Security conference, Anibal Sacco and Alfredo Ortega demonstrated patching malicious code into the decompression routines of the BIOS.[5] Similar to Heasman's finding, physical access and reflashing capability is required to mount the attack.

Requiring physical access and reflashing BIOS firmware with an attacker's code significantly limits the value of the proposed attacks, because nowadays, most manufacturers do not allow arbitrary programming of the BIOS. When manufacturers issue BIOS updates for adding hardware support and fixing bugs, the new BIOS images are usually digitally signed with the manufacturer's private key. Only if the signature checks out by the operating system will the BIOS update be scheduled to launch after reboot.

At the Black Hat USA conference in 2009, Rafal Wojtczuk and Alexander Tereshkin presented an attack against certain vulnerable BIOS.[6] The attack exploits a buffer overflow bug in these BIOSes to subvert the integrity protection (digital signature) on the BIOS update. The attack is more sophisticated than the ones introduced by Heasman, Sacco, and Ortega, because it does not require physical access, making remote and wide deployment possible.

Software Replacement

Attacks can be classified into various models according to the intension. With the exception of the CIH virus, the attacks discussed so far in this chapter target taking control of victims' computers and stealing secrets or performing other harmful operations.

In other models, however, attackers are playing with and hacking their owner devices, in the attempt to achieve certain goals:

- *Install adversary's software system on a low-end device*: The software shipped with low-end hardware by its OEM (original equipment manufacturer) may come with limited functionalities. It is to the user's interest to replace the original software stack with unauthorized software, where more powerful functionalities are available; for example, installing Android on a GPS (Global Positioning System) or media player device. Notice that the low-end device may not be equipped with premium hardware features, which limits what the adversary's software is able to accomplish.

- *Install adversary's software system on a high-end device*: The high-end device features hardware capabilities to support premium functionalities, such as enhanced high-definition movie playback, near field communication (NFC), and so forth. The adversary's software can bypass certain restrictions. For example, content protection may be deployed by an OEM's software to enforce a movie rental period. The adversary's software may remove such policy so that the user can own the movie permanently.

Jailbreaking

Jailbreaking or *rooting* refers to the action of overcoming certain restrictions of the firmware and software stack that are installed on the device by the device OEM or carrier (in the case of a smartphone). Essentially, jailbreaking is a form of privilege escalation that allows the user to gain the root privilege and full control of his device.

It is common practice for OEMs and wireless carriers to implement restrictions in the firmware and software that is shipped with the hardware. There are a number of reasons for this practice. For example, here are a few:

- Selling applications and additional services to users after they purchase the device
- Protecting the device from malware and viruses
- Promoting the OEM's software products by preinstalling and locking them down in the operating system
- Preventing the wireless device under service contract from being used with other carriers
- Collecting usage data from wireless subscribers

Jailbreaking would invalidate all aforementioned purposes; hence it is against the OEM and carrier's interest. For example, a jailbroken iPhone or iPad may be able to run third-party applications that are not authorized by or purchased from the official Apple App Store. It is also possible to jailbreak a smartphone, unlock premium services, and enjoy them for free, while the carrier intended to collect extra charges for these services. For example, tethering or Hotspot is usually a paid function charged by the amount of 4G data shared between the smartphone and other non-4G platforms, such as a laptop. Software of a jailbroken phone may cheat the carrier by reporting tethering or Hotspot traffic as regular 4G data, hence avoiding extra charges.

Besides circumventing restrictions in the existing firmware and software stack, a more sophisticated form of jailbreaking is to install a completely different software system and possibly repurpose the device. This is especially interesting for devices that are equipped with powerful hardware capabilities but limited software functionalities. HP's TouchPad is such an example.

Launched in July 2011, the TouchPad was discontinued less than two months later. Remaining inventories were sold at extremely low prices to clear the stock. The TouchPad was made of state-of-the-art hardware specifications for that time, including a 1024×768–pixel touch screen, 16GB or 32GB of storage, and 1GB of memory. The operating system preinstalled on the TouchPad was the webOS, which suffers several limitations, such as very small number of available apps, compared to its competitors, iOS and Android. Obviously, it is to the users' interest if a "better" operating system can be installed to run on the TouchPad hardware. In October 2011, the first Android-based jailbreak was released by CyanogenMod.[i] The CyanogenMod converts the TouchPad to a dual-boot system that supports both webOS and Android.

[i]CyanogenMod is a free open source operating system for smartphones and tablets, based on the Android mobile platform.

In most cases, jailbreaking is made possible by exploiting design flaws or vulnerabilities in the firmware or software. For example, if a manufacturer's firmware is not digitally signed, then it is convenient to replace it with an adversary's firmware. Even if the architecture and design are sound, bugs in implementation may be exploited to allow jailbreaking.

Now, when the device owner is the hacker, how does the device protect itself from being broken? Clearly, a meaningful integrity protection scheme would have to depend on a root of trust that is in hardware and intact from alteration. How do Intel's CPU and security and management engine help with this matter?

Trusted Platform Module (TPM)

Discussions regarding the integrity of firmware and software on a platform always involve trusted platform module[7] (TPM). The TPM is a public standard that defines the interfaces of a security coprocessor. A TPM implementation is a hardware device that provides cryptographic functionalities for the software to invoke.

Because the TPM is hardware, it is more difficult for attackers to break its security and protections. Attacks against hardware are usually attempted through side channel analysis; for example, timing information, power consumption, and electromagnetic emissions. These attacks require not only physical access, but also special equipment and advanced skills. These requirements limit the scope of the damage of successful attacks, because the hardware attacks cannot be reproduced widely and easily by spreading viruses or malware.

Beside its hardware nature, another important feature of the TPM is its independence. The TPM is a module isolated from the main operating system. Its operations do not rely on and is not impacted by the operating system or the software running on it. This makes the TPM a trustworthy "third-party" for examining the integrity of the software stack.

TPM may be implemented as a physically discrete device or as a logical component inside a security coprocessor. Recent generations of Intel's secure and management engine features a firmware TPM, which is used to support secure boot designs as well as other purposes defined in the TPM standard. For more information about the TPM on the embedded engine, refer to Chapter 7 of this book. Despite the existence of the firmware TPM, it is also possible to include a discrete TPM in the platform. Intel's secure boot architecture, Intel Boot Guard, can work with either the firmware TPM or a discrete TPM.

Platform Configuration Register

The primary goal of the TPM is to protect the integrity of the platform. As such, it is equipped with implementations of hash algorithms and one or more banks of platform configuration registers (PCRs). During the boot process, the PCRs can be used to store and report the hash results for every firmware and software component. The operation of hashing a boot component is often referred to as a *measure*. The operation of measuring the next component is often referred to as an *extend*, because the measurement of the next component is against not only the next component, but also all components that have been measured before it. In other words, the measurement is always incremental. This is defined in the following formula:

$$\text{digest}_{\text{new}} := H_{\text{hashAlg}}\left(\text{digest}_{\text{old}} \| \text{data}_{\text{new}}\right)$$

In this formula, || means concatenation and $\mathbf{data}_{\text{new}}$ refers to the binary data of the component being measured. $\mathbf{H}_{\text{hashAlg}}$ is the chosen hash algorithm, like SHA-256. From the formula, it is easy to understand that an altered component that is loaded during the boot process will result in incorrect or unexpected measurements for not only itself, but also all components loaded after it, even though those components are intact. Typically, the measurements are checked later locally or reported to remote servers for attestation. The TPM serves as secure storage only and does not perform the comparison for measurements.

Notice the PCR is not specific for the boot time measurement. Rather, supporting the integrity of boot components is just one of many usage models of the PCR. Per the TPM specification, the PCRs are designed for generalized representation of a platform state, and platform-specific specifications may define additional PCR behaviors. In general, a platform specification may define a PCR to represent any value that is authoritatively known by the TPM or has been securely communicated to the TPM.

Many secure boot architectures take advantage of the TPM's measurement capability. However, the TPM has other useful ingredients in addition to the PCR, and the TPM is not just about protecting boot integrity. The TPM has a range of cryptographic capabilities, such as sealing and binding data, to help secure the platform not only during boot but also at runtime.

Field Programmable Fuses

Newer security and management engines shipped with select Intel platforms in and after 2013 support a feature called *field programmable fuses*. As its name indicates, it allows fuses to be burned after leaving Intel's manufacturing facility, in the OEM's factory or in the field. The field programmable fuses are essentially another nonvolatile storage medium. However, it is not the only nonvolatile storage in the engine.

Field Programmable Fuses vs. Flash Storage

The security and management engine's kernel contains a storage manager that manages nonvolatile data that must persist across power cycles. Nonsensitive data can be stored in plaintext; secrets can be protected with confidentiality, integrity, and anti-replay. The embedded applications that invoke the storage manager are free to apply one or more of these protection options for their data. The data is stored on the flash device in a special partition. The same flash also stores the BIOS, the embedded engine's binary image, as well as other system firmware.

Now that nonvolatile data can be stored on the flash, why the field programmable fuses? When comparing the field programmable fuses with the flash storage, anti-replay becomes an interesting aspect. Two anti-replay mechanisms are supported by the storage manager: native monotonic counter and RPMC (replay-protected monotonic counter) flash:

- *Native monotonic counter*: The monotonic counter resides in the chipset's RTC (runtime clock) power well. Upon RTC power loss, for example, due to coin battery removal, all anti-replay blobs managed by the engine are invalidated by the storage manager. Because of this limitation, the applications must be able to re-create the blobs in case they are lost.
- *RPMC flash*: The flash device natively mitigates anti-replay attacks. The advantage is the independence of the RTC power well. The disadvantage is the cost of the RPMC flash. Not all OEMs use RPMC flash parts for all products.

The field programmable fuse scheme provides anti-replay protection that completely eliminates the dependency on RTC well or RPMC flash. Thanks to its nature, writing a fuse is a one-time operation. That is, once a fuse has been burned (its value changing from 0 to 1), the operation cannot be reversed, and the fuse will assume the value of 1 from then on. This characteristic makes field programmable fuses especially suitable for holding data that requires certain properties:

- The data must survive flash wipe or corruption. Such data includes platform state information, OEM programmable confidential information, and so forth. The security and management engine's verified boot architecture uses the field programmable fuses for OEMs to program digests of their public keys.
- The data is used to support security claims; loss of the data may result in security vulnerabilities. For example, the fuses can be used to permanently record the fact that a security enhancement feature, such as anti-theft or TPM, has been enabled for this platform. If an attacker (owner of the device) intends to bypass specific restrictions by reflashing the firmware image with another version that does not support the security enhancement, then the image replacement will be caught by the fuses.

The storage manager is not able to provide this level of protection with its anti-replay mechanisms.

In addition to anti-replay, the fuse block is hidden inside the security and management engine and invisible to the outside of the engine. In other words, confidentiality and integrity are native characteristics of the field programmable fuses, without having to apply encryption and hashing algorithms.

The main drawback of field programmable fuses is the relatively small number of fuses available on die. For a typical configuration of the engine, there are 1024 programmable fuses in a 32×32 array layout. About one in every four fuses is reserved for locking, repairing, and redundancy check purposes, leaving only a few hundred fuses for applications to program. As such, the uses of the field programmable fuses are not a runtime matter, and must be predefined and allocated carefully on a case-by-case basis.

Field Programmable Fuse Task

From the firmware architecture perspective, the field programmable fuse manager is implemented in its own *task* (container). See Chapter 4 of this book for more information about the security and management engine's task isolation infrastructure. Being a dedicated task, other tasks are not able to penetrate the field programmable fuses. Firmware modules that own fuses can program or sense the fuses by calling the field programmable fuse task via the intertask calling mechanism supported by the kernel.

The flow for programming a fuse is depicted in Figure 6-2. The figure does not detail steps for the fuse manager to burn a fuse; for example, a valid bit check, a redundancy check, and so forth. The flow for sensing the value of a fuse is similar and is not shown in this figure.

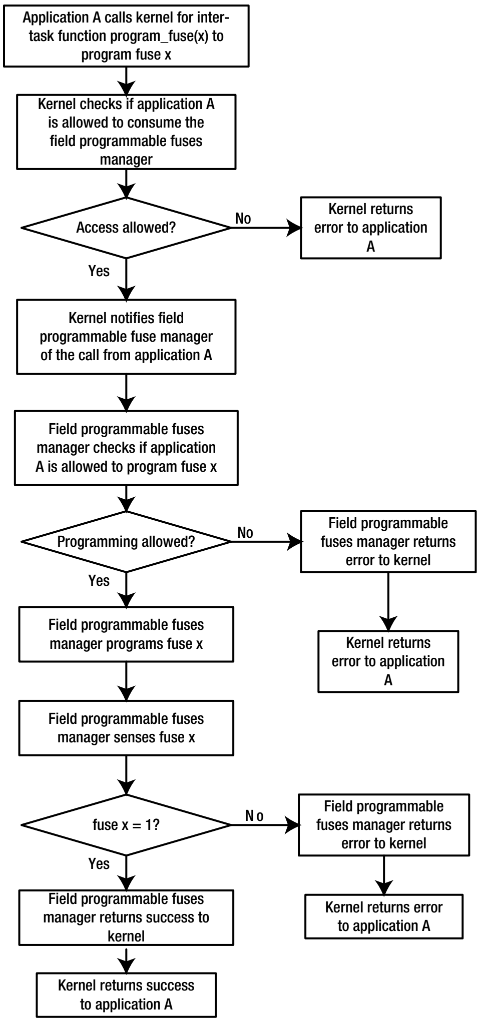
Application A calls kernel for inter-task function program_fuse(x) to program fuse x
Kernel checks if application A is allowed to consume the field programmable fuses manager
Access allowed?
No
Kernel returns error to application A
Yes
Kernel notifies field programmable fuse manager of the call from application A
Field programmable fuses manager checks if application A is allowed to program fuse x
Programming allowed?
No
Field programmable fuses manager returns error to kernel
Kernel returns error to application A
Yes
Field programmable fuses manager programs fuse x
Field programmable fuses manager senses fuse x
fuse x = 1?
No
Field programmable fuses manager returns error to kernel
Kernel returns error to application A
Yes
Field programmable fuses manager returns success to kernel
Kernel returns success to application A

Figure 6-2. *Flow for application A to programming fuse x*

Depending on the nature of the data, there are five usage models of the programmable fuses:

- *Single-bit one-time programming*: The data is of Boolean type. Once programmed, the change becomes permanent and it can no longer be reverted. This usage requires only one fuse. For example, once the OEM finishes the manufacturing process, it programs a single-bit one-time fuse to show that manufacturing is completed. Certain configurations of the security and management engine are intended for only OEMs to use; it is not supposed to be touched by end users. The firmware logic for handling such configurations consults this "end of manufacturing" indicator fuse, before proceeding with the configuration manipulation.
- *Single-bit multiple-time programming*: The data is of Boolean type. It may change a limited number of times, say *n*, during the lifetime of the platform. In this case, *n* fuses are necessary for storing the data, and one of the *n* fuses is programmed every time the value of the data flips. Take the anti-theft technology for example. Once enrolled, the anti-theft technology automatically shuts down the platform per the user-configured policy if it detects that the system is in a stolen state. The shutdown is performed only if the platform is enrolled, therefore the enrollment status is critical for enforcing the shutdown. Users are free to opt out after enrollment or enroll again (that is, changing the enrollment status) for a limited number of times. For a single-bit multiple-time programming fuse, the field programmable fuse manager counts the number of the *n* fuses that have been burned. If the number is odd, then the data is assumed to be *true*, implying, for example, the anti-theft is currently enrolled; if the number of is even or zero, then the data bit is assumed a value of *false*.
- *Multiple-bit one-time programming*: The data consists of multiple bits. It cannot be changed once programmed. For this usage, the number of fuses required is equal to the bit size of the data. For example, in Intel's verified boot architecture, the OEM programs its 256-bit hash of OEM's RSA public key to field programmable fuses during the manufacturing process. The OEM also programs its secure boot policies to designated fuses. Once done, the value cannot be erased or updated during the lifetime of the platform.
- *Multiple-bit multiple-time programming*: The data consists of multiple bits. It may change for a limited number of times. For this usage, the number of fuses required is equal to the data's bit size multiplied by the number of times the data is allowed to change during the lifetime of the platform.

- *Incremental integer*: The data is a non-negative integer that assumes values from 0 to *m*, inclusive. The data assumes an initial value of 0 and can only be updated from smaller to greater; for example, from 1 to 3, but not from 3 to 2. A set of *m* fuses are required for this usage model. The number of burned fuses represents the value of the data. A typical usage is the version number of a firmware component. When vulnerabilities are fixed in a firmware patch, the version number of the new release will be incremented by one from the previous vulnerable version. The latest version number is recorded in the fuses. When the embedded engine loads the firmware, it checks the firmware's version number and compares with what is shown by the fuses. If the former is greater, the fuses are updated with the new version number; if the former is smaller, then the system concludes that it is under a rollback attack and proceeds accordingly.

Intel Boot Guard

Intel Boot Guard technology provides hardware-based boot integrity protection that prevents malicious firmware and software from taking over boot blocks. It does so by detecting an unauthorized boot block and disallowing it to execute. The Boot Guard is a hardware and firmware solution that does not depend on any software.

Intel released the authenticated code module, or ACM, for OEMs to enable the Intel Trusted Execution Technology[8] (TXT) and the Boot Guard feature. As will be described later in detail, the ACM plays a pivotal role and carries critical tasks in the Boot Guard solution. Digitally signed by Intel, the ACM component is stored on the flash together with BIOS and other firmware components. The public key for verifying the signature on the ACM is hard-coded in Intel's CPU. There is a security version number associated with the ACM module, which is used to identify and revoke vulnerable ACM releases and stop the system from booting.

To take advantage of the Boot Guard technology, the OEM must implement a new firmware component to the boot flow, called the *initial boot block*, which is loaded before the BIOS. The initial boot block is responsible for checking the integrity of BIOS, initializing memory, and loading BIOS into the system memory. Just like the ACM, the initial boot block is stored on the flash chip. The boot flow is shown in Figure 6-1, and the additions of the ACM and the initial boot block are shown in Figure 6-3.

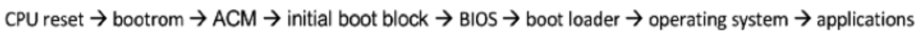

Figure 6-3. *Boot flow with ACM and initial boot block*

Note that this is a simplified boot flow. The boot flow with the TXT is more complicated. Intel Boot Guard technology defines three boot configurations:

- *Measured boot*: Measures the initial boot block into the platform's secure storage device, such as a TPM.
- *Verified boot:* Cryptographically verifies the integrity of the initial boot block using a digital signature scheme. The verified boot reduces material cost because it offers boot protection without a TPM device.
- *Measured boot + verified boot*: Measures and verifies the initial boot block.

But, why is it necessary to introduce the initial boot block? Why can't the Boot Guard directly verify the BIOS? Here are a couple reasons.

- *Size*: The size of today's BIOS image is in the scale of megabytes and increasing. However, the initial boot block is desired to be small enough to fit in the on-die memory of Intel silicon in all compatible platforms. In other words, the architecture must work with fixed and limited memory size. This is not scalable for a BIOS whose size may increase.
- *Flexibility*: Modularity in design provides flexibility and the ease of changing only parts of the product. Also, an OEM can use one private key to sign the initial boot block and another key to sign the BIOS. Even in the event the private key for signing the BIOS image is leaked or compromised, there is no need to recall hardware.

Operating System Requirements for Boot Integrity

Microsoft's Windows Certification Program[9] specifies a requirement for boot integrity. Intel's Boot Guard technology helps OEMs meet this requirement for their Windows-based systems:

> *Boot Integrity: Platform uses on-die ROM[ii] or One-Time Programmable (OTP) memory for storing initial boot code and initial public key (or hash of initial public key) used to provide boot integrity, and provides power-on reset logic to execute from on-die ROM or secure on-die SRAM.[iii]*

Google does not pose requirements for boot integrity for Android-based systems. In fact, most Android device manufacturers do not implement a secure boot, and intentionally allow a custom operating system to be loaded.[10] CyanogenMod is one of the most famous customized mobile operating systems derived from Android. Tutorials and materials for rooting Android devices are publicly available.

[ii]Read-only memory

[iii]Static random-access memory

OEM Configuration

The Boot Guard configurations set by the OEM slightly vary among different products. In general and at a minimum, the OEM is responsible for configuring its public key hash for a verified boot, and the boot policies via the security and management engine.

The security of a verified boot is rooted to the OEM's asymmetric keypair. The OEM generates a 2048-bit RSA keypair as its root key for signing manifests for the initial boot blocks. The private portion of the root keypair must be kept securely, and signing manifests for initial boot blocks shall be its sole usage. On the other hand, the SHA-256 hash of the public key is programmed to the field programmable fuses during the manufacturing process. The public key hash consumes 256 fuses that belong to the *multiple-bit one-time programming* category, which cannot be updated once written. Because of the one-time programming limitation, the OEM will not be able to renew the root key or update the hash, even if the private key is compromised. Therefore, the OEM must protect its root private key in a signing server with strong protection from attacks or leakage.

In addition to programming its public key hash, the OEM is also responsible for defining its boot policies and saving them in the field programmable fuses. The boot policies are also a one-time configuration that cannot be revised. The policies instruct the Intel hardware with regard to the following:

- What boot protections are enabled—that is, measured boot only, verified boot only, neither, or both
- What actions to take upon ACM failure
- What action to take upon initial boot block failure

In the scenario that the CPU is unable to load the ACM from the flash or the digital signature of the ACM fails to verify, the CPU may either (based on the OEM's setting for the second bullet in the preceding list) enter the shutdown state or proceed with booting from the legacy vector. Although the instant shutdown option offers the highest level of integrity protection, it is generally not recommended because it may potentially lead to a large number of customer support calls. And problems are extremely difficult to debug if the system powers itself off at a very early stage of the boot process.

After the ACM is checked out successfully, the initial boot block becomes the next subject of interest. Recall that the security and management engine is capable of triggering instant shutdown of the platform (see Chapter 4 for details). When a boot integrity-check fails, it is the engine's responsibility—according to the OEM's set policies—to shut down the platform and terminate the boot process. The OEM can determine when the shutdown should happen upon failure. A few options are available:

- *Unrestricted*: Do not shut down the system; let it boot and run normally as if the failure did not occur.
- *Remediation*: Let the system continue to boot but shutdown ungracefully after a certain amount of time. The amount of time (for example, 30 minutes) should be enough for a repair technician to perform basic remediation work, such as updating the initial boot block or BIOS from the operating system. Yet, the time before shutdown should not be too long; otherwise, the boot policy becomes meaningless.

- *Diagnostics*: This is similar to the remediation option, but the timer is set to a much smaller value, such as one minute. This option allows the manufacturer's support engineers to retrieve debug information from the system.
- *Zero-tolerance*: Shut down the platform immediately upon a boot integrity failure. Similar to the case of ACM failure, this option is generally not recommended.

The security and management engine offers two methods for the OEM to program its public key hash and the boot policies to the designated field programmable fuses. In both cases, the configuration is allowed only before the end of the manufacturing process:

1. *Through HECI*[iv] *commands sent from the host operating system*. The commands are honored by the engine only before the "end of manufacturing" HECI message is received and recorded. This method is not available for production parts.
2. *Through image building*. Intel provides OEMs with a software program called *firmware image tool* to build a flash image from various components, such as binaries of BIOS, the security and management engine, and so on. The tool allows an OEM to configure the engine for Boot Guard support, including setting its public key hash and boot polices. These values will be automatically programmed to the field programmable fuses by the engine's firmware as soon as the "end of manufacturing" HECI message is received and recorded.

The boot policy configuration applies to both the measured boot and verified boot.

Measured Boot

The measured boot mechanism is made possible by the Intel TXT. The Intel TXT is designed to harden platforms at the hardware level, from hypervisor, firmware (BIOS, root kit, and so forth), and other software-based attacks.

The Windows Certification Program requires measuring all boot components using a TPM. Intel's measured boot meets this requirement because the initial boot block is measured as the first boot component:

> *During the boot sequence, the boot firmware/software shall measure all firmware and all software components it loads after the core root of trust for measurement is established. The measurements shall be logged as well as extended to platform configuration registers in a manner compliant with the following requirements.*

[iv]HECI, or host-embedded communication interface, is the two-way communication channel between the security and management engine and the host operating system. Refer to Chapter 3 for more information about the HECI.

The Intel TXT works by creating a measured launched environment (MLE), which enables precise comparisons between the current state of the platform and known-good references for all components of the boot process. The measurements (extended hashes of components) are stored in the platform's secure storage device, usually a TPM, and are available for local or remote attestation. If measurements match known-good configurations, then the TXT marks the system *trusted*; otherwise, the TXT marks the system *untrusted* and follows defined fallback policies. It can either abort the boot process or let the platform continue to operate—but with degraded functionality, such as forbidding it from running sensitive tasks, for example.

For the measured boot, the CPU loads the ACM after verifying the signature associated with it. The ACM calculates the hash of the initial boot block and stores the measurement in a PCR slot of the platform's discrete or firmware TPM device. The measurement is available for attestation later.

Verified Boot

The measured boot mechanism relies on a dedicated storage device, typically PCR slots of a TPM, to securely store measurements of the initial boot block and other components involved in the boot process. Unfortunately, a TPM may not be available on all form factors. This is especially the case for low-cost mobile devices. Specifically, for systems in which TPM is not required for other functionalities, adding a TPM merely for the purpose of safeguarding the boot integrity increases not only the BOM (bill of materials) cost but also development and integration effort, which may not yield a good return-on-investment. However, the boot integrity can still be a critical requirement for those devices. The verified boot mechanism provides an alternate approach without relying on a TPM or other devices. Notice that the verified boot mechanism by itself does not measure all boot components. Therefore, without a measured boot, it may not satisfy the Windows Certification Program requirements.

Cryptographically, data integrity is achieved by employing either a hash (including a keyed hash and a plain hash) or a digital signature as a "measurement." Without an independent and trusted reference, the "known good" measurement must be kept within the platform and intact from unauthorized alteration. The verified boot features a hardware-based root of trust for verifying the integrity of the initial boot block. Next, the initial boot block verifies the integrity of the BIOS, the BIOS verifies the integrity of the boot loader, and the boot loader verifies the integrity of the operating system, and so forth. The integrity of successive components loaded following the initial boot block is guaranteed by a chain of trust.

Manifests

The initial boot block binary is associated with a manifest, called the *initial boot block manifest*, or IBBM for short. The IBBM contains the following fields:

1. The security version number of the IBBM
2. The SHA-256 hash of the initial boot block

3. The RSA signature on (1) and (2)

4. The RSA public key that is used to verify (3), referred to as the *IBBM public key* onward

The IBBM 2048-bit RSA keypair is also generated by the OEM, but it is different from the OEM root RSA keypair introduced earlier; although an OEM is free (but not encouraged) to utilize the same keypair for both. The only usage of the IBBM RSA keypair is to sign IBBMs. The IBBM RSA private key must be kept securely by the OEM. The OEM root public key hash is stored in the security and management engine's programmable fuses. In contrast, the IBBM public key appears only in the IBBM.

The IBBM is not the only manifest in the picture. The OEM uses its root keypair to sign another manifest, namely the *key manifest*, which contains the following fields:

5. The security version number of the key manifest

6. The SHA-256 hash of the IBBM public key

7. The RSA signature on the (5) and (6)

8. The OEM root public key, used to verify (7)

The hash of the OEM root public key (8) is stored in the programmable fuses. Both the IBBM and the key manifest are stored on the flash. The relationships among the root key hash, two manifests, and the initial boot block are better explained graphically in Figure 6-4.

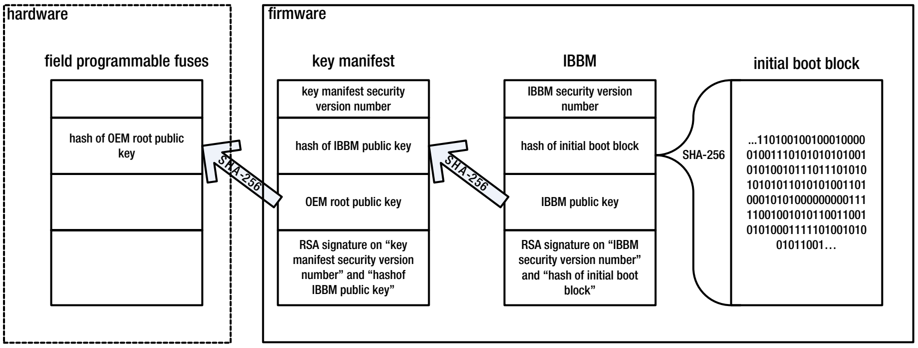

Figure 6-4. *Using the OEM public key hash to verify the initial boot block via the key manifest and IBBM*

As Figure 6-4 depicts, the root of trust is the OEM root public key hash located in the fuse hardware and handled by the security and management engine. This makes the verified boot a hardware-based scheme that is significantly more difficult to compromise than software solutions.

The key manifest seems an unnecessary middleman sitting between the OEM root public key hash and the IBBM. Why not just use the OEM root key to sign the IBBM directly? The indirection introduced by the key manifest is desirable for OEMs that manufacture multiple product lines. With the key manifest, the OEM can use a single root key for all its products, but different IBBM keys for different product lines.

For the sake of revocation, both manifests are versioned.

- The security version number of the key manifest enables the OEM to revoke the IBBM keypair should it be compromised. If the IBBM keypair must be replaced, then the OEM will generate a new IBBM keypair and place its public key hash in a new key manifest, and at the same time increment the security version number of the key manifest.
- The security version number of the IBBM covers the initial boot block, and it allows the OEM to revoke and patch a vulnerable initial boot block. When a new initial boot block is released, the security version number of the IBBM must be incremented accordingly.

The two version numbers are examined by the security and management engine during the verified boot process. If the engine finds that the version number of a manifest being loaded is greater than the corresponding value recorded in the field programmable fuses, then it programs a certain number of fuses to reflect the greater version number. The fuses reserved for the security version numbers belong to the category of *incremental integer*. The version number of a manifest being loaded being smaller than the corresponding value recorded in the fuses is an indicator of a rollback attack, where an attacker unlocks the flash part and replaces a good and later version of the manifest with a vulnerable and older version. In this situation, the embedded engine will react accordingly per the boot policies in the fuses configured by the OEM.

Admittedly, revocation relying on security version numbers has its limitations. The mechanism works only if the platform has already run, at least once, a later manifest or an initial boot block with a greater version number, and then the manifest or initial boot block is rolled back to an earlier and vulnerable version. If the attacker blocks manifest or initial boot block updates (this is rather trivial to do) in the first place, so the platform has no chance to ever see the patched manifest or initial boot block, then the revocation design backed by security versioning will not be able to protect the platform. To make the situation worse, an advanced attacker may reverse-engineer the new initial boot block release and figure out the security bugs that were fixed, and attempt to exploit the bugs in the old initial boot block.

Verification Flow

The verification of the initial boot block is a collaborative effort by the security and management engine and the ACM running on the CPU. The ACM is responsible for the following:

- Loading the initial boot block firmware and the two manifests from the flash
- Retrieving the OEM's public key hash, boot policy, its own security version number, and the security version numbers of the two manifests from the engine

- Verifying the integrity of the initial boot block using the manifests and OEM's public key hash
- Notifying the engine of updating the security version numbers if necessary
- Enforcing boot policy in the event of a communication error or a time-out with the engine

The security and management engine is responsible for the following:

- Reading OEM's public key hash, boot policy, ACM security version number, and the security version numbers of the two manifests from field programmable fuses, and sends to the ACM
- Incrementing security version numbers of the ACM and the two manifests in the fuses upon requests from the ACM
- Enforcing boot policies in the event of a communication error or time-out with the ACM
- Performing appropriate actions upon failure of verification, per the boot policies

Figure 6-5 presents the high-level sequence diagram. In the figure, the security version number check performed by the ACM is against three elements: the ACM, the key manifest, and the IBBM. For the boot process to succeed, all three values seen by the ACM must be equal to or greater than the respectively referenced values reported by the security and management engine. If one or more of the security version numbers need updating, then the ACM notifies the engine after all checks have passed.

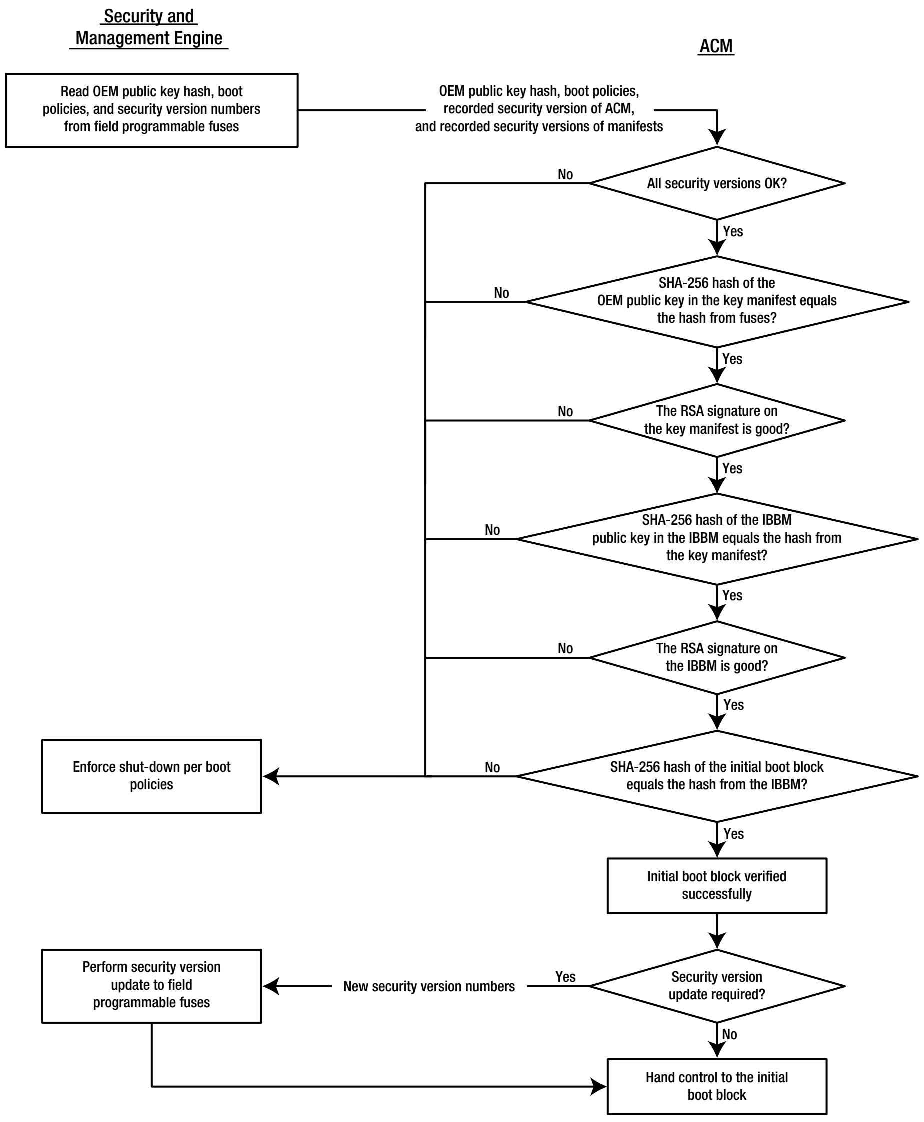
Security and Management Engine
ACM
Read OEM public key hash, boot policies, and security version numbers from field programmable fuses
OEM public key hash, boot policies, recorded security version of ACM, and recorded security versions of manifests
All security versions OK?
No
Yes
SHA-256 hash of the OEM public key in the key manifest equals the hash from fuses?
No
Yes
The RSA signature on the key manifest is good?
No
Yes
SHA-256 hash of the IBBM public key in the IBBM equals the hash from the key manifest?
No
Yes
The RSA signature on the IBBM is good?
No
Yes
SHA-256 hash of the initial boot block equals the hash from the IBBM?
No
Enforce shut-down per boot policies
Yes
Initial boot block verified successfully
Security version update required?
Yes
New security version numbers
Perform security version update to field programmable fuses
No
Hand control to the initial boot block

Figure 6-5. *The initial boot block verification flow for the verified boot*

References

1. Joanna Rutkowska, "Evil Maid Goes After TrueCrypt," http://theinvisiblethings.blogspot.com/2009/10/evil-maid-goes-after-truecrypt.html, accessed on March 20, 2014.
2. J. Alex Halderman, Seth D. Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A. Calandrino, Ariel J. Feldman, Jacob Appelbaum, and Edward W. Felten, "Lest We Remember: Cold Boot Attacks on Encryption Keys," *Proc. 17th USENIX Security Symposium*, San Jose, CA, July 2008.
3. Unified EFI, Inc., "Unified Extensible Firmware Interface Specification," www.uefi.org, accessed on March 20, 2014.
4. John Heasman, "Implementing and Detecting an ACPI BIOS Rootkit," Black Hat Europe, March 3, 2006, Amsterdam, the Netherlands.
5. Anibal Sacco and Alfredo Ortega, "Persistent BIOS Infection," CanSecWest, March 19, 2009, Vancouver, BC.
6. Rafal Wojtczuk and Alexander Tereshkin, "Attacking Intel® BIOS," Black Hat USA, July 30, 2009, Las Vegas, NV.
7. Trusted Computing Group, "Trusted Platform Module Library," www.trustedcomputinggroup.org, accessed on March 20, 2014.
8. Intel Trusted Execution Technology, www.intel.com/txt, accessed on January 30, 2014.
9. Microsoft Corporation, *"Windows Certification Program: Hardware Certification Taxonomy & Requirements—Systems,"* December 16, 2013, pp. 125.
10. N. Asokan, Lucas Davi, Alexandra Dmitrienko, Stephan Heuser, Kari Kostiainen, Elena Reshetova, and Ahmad-Reza Sadeghi, *"Mobile Platform Security,"* Morgan & Claypool Publishers, 2013, pp. 40.